

A VIC-20 COMPILER FOR CONTROL SYSTEM

by

Oanh Kim Ngoc Huynh

B.A. in Comp. Sc. & Math., University of Pittsburgh

Submitted to the Graduate Faculty

of the School of Engineering

in partial fulfillment of

the requirements for the degree of

Master of Science

in

Electrical Engineering

University of Pittsburgh

1985

The author grants permission
to reproduce single copies.

Oanh K. N. Huynh
signed

ACKNOWLEDGEMENT

I am greatly indebted to many people for their timely help and valuable suggestions. In particular, I wish to express my gratitude to my major advisor, Dr. W. G. Vogt for suggesting the topic, for his tireless assistance, guidance and encouragement throughout the preparation of this thesis.

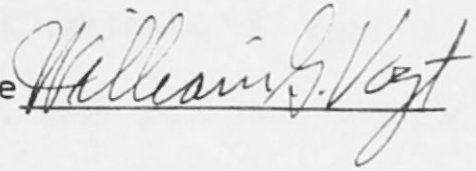
I also would like to extend my appreciation to the members of the oral examination committee Dr. T. W. Sze, Dr. M. H. Mickle and Dr. W. G. Vogt for their help, guidance and many kindnesses, for ensuring the clarity and accuracy of my thesis.

Special thanks are due to my sister and brother in law, Thu and Phong Do, who spent so much of their valuable time to draw the figures and make production of the entire manuscript a number of times. And last but not the least to Khoan, my husband, for many hours I spent with him discussing various aspects of this study, for practical

suggestions and help, and for providing valuable and otherwise unavailable information.

ABSTRACT

Signature



A VIC-20 COMPILER FOR CONTROL SYSTEM

Oanh Kim Ngoc Huynh, M.S.

University of Pittsburgh, 1985

The implementation of a real time computer controller on a personal computer has a significant value in today's technology. For example, taking advantage of the existing hardware and accessories of a VIC-20, any simple real time control system can be realized through the help of the Basic Interpreter built into the VIC-20. This is not always true when it comes to the point where speed is critical and multiple tasks are required. Then, to be able handle multitask programs, an expensive computer is required; to be able to handle time critical tasks, a better and faster execution time is required.

This thesis proposes an intermediate way to achieve both goals with the same minimum hardware possible : a VIC-20 and its accessories. Instead of writing a control algorithm through the basic interpreter, one can accomplish the work directly by use of a machine language program, which is generated by a new Basic compiler described in this thesis. This compiler has special functions for computer control applications,

DESCRIPTORS

Computer Control	Digital Control
Compiler	Stepper Motors
DC Motors	VIC-20

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENT	ii
ABSTRACT.	iv
TABLE OF CONTENTS	vi
LIST OF FIGURES	x
SCOPE OF THE THESIS	1
1.0 INTRODUCTION	5
1.1 BACKGROUND.	5
1.2 VIC-20 AND ACCESSORIES.	6
1.3 BASIC INTERPRETER OF VIC-20	7
1.3.1 VIC USER MEMORY.	7
1.3.2 I/O OPERATION.	10
1.3.3 SUBROUTINE CALLING	10
1.3.4 SPEED.	12
2.0 TINY BASIC COMPILER.	14
2.1 EDITING TEXT --- VIC-20 ---	14
2.2 HOW DOES THE TINY BASIC COMPILER WORK ?	16
2.2.1 GRAMMAR.	18
2.2.2 CODE GENERATION	22

3.0	SYSTEM OVERVIEW.	25
3.1	HARDWARE (SYSTEM REQUIREMENTS).	25
3.2	SOFTWARE (SYSTEM STRUCTURE)	29
3.2.1	I/O CONTROL SUBROUTINE ADDRESSES	33
3.3	I/O MEMORY MAP FOR CONTROL LANGUAGE	35
3.3.1	INPUT/OUTPUT AND VIA INTERFACE	35
3.3.2	STEPPER MOTOR AND VIA INTERFACE.	36
3.3.3	DC MOTORS AND VIA INTERFACE.	38
3.3.4	THE LOCAL CONTROL TABLE FOR I/O ROUTINES	41
3.3.5	NECESSARY SUBROUTINES FOR PASSING FLOATING ACCUMULATOR	42
4.0	SYSTEM ARCHITECTURE AND CAPACITY	43
4.1	CONTROL LANGUAGE.	43
4.1.1	GRAMMAR OF THE CONTROL LANGUAGE.	43
4.1.2	SYNTHESIS.	55
4.1.2.1	WORD IDENTIFICATION	55
4.1.2.2	CODE GENERATION	57
4.1.2.3	DETAILED DESCRIPTION OF THE CONTROL LANGUAGE.	59
4.2	CAPACITY.	74
4.2.1	MAX-MIN STORAGE REQUIREMENT.	74

4.2.2	I/O CAPACITY	75
4.3	LOADER'S SPECIFICATION.	76
4.3.1	MEMORY REQUIREMENT	77
4.3.2	I/O PRIMITIVES --- RESERVED MEMORY LOCATIONS	77
4.3.3	I/O PRIMITIVE DESCRIPTION.	78
4.4	LOADING AND EXECUTION	79
5.0	APPLICATION.	80
5.1	OPEN LOOP REAL TIME CONTROL	81
5.1.1	SAMPLE RUN	81
5.1.2	EFFICIENCY AND EXPANSION	82
5.2	CLOSED LOOP REAL TIME CONTROL	83
5.2.1	SAMPLE RUN	83
5.2.2	EFFICIENCY AND EXPANSION	87
5.3	MULTITASKING.	88
5.3.1	SAMPLE RUN	89
5.3.2	EFFICIENCY AND EXPANSION	90
6.0	CONCLUSIONS.	91
APPENDIX A.	96
A.1	I/O BUS SPECIFICATIONS.	96
A.2	UNIVERSAL INTERFACE MODULE.	97

A.3	STEPPER MOTOR INTERFACE (INPUTS)	100
A.4	STEPPER MOTOR INTERFACE (OUTPUTS)	103
A.5	DC MOTOR INTERFACE MODULE	103
A.6	ANALOG INPUTS INTERFACE MODULE.	111
A.7	STEPPER MOTOR DRIVER MODULE A/B	112
A.8	DC MOTOR DRIVER MODULE.	115
A.9	SPEED SENSING MODULE.	120
A.10	INPUT CONDITIONING MODULE.	123
	REFERENCES NOT CITED.	128

Figure A-3:	UNIVERSAL INTERFACE MODULE.	101
Figure A-4:	UNIVERSAL INTERFACE SPECIFICATION .	102
Figure A-5:	STEPPER MOTOR INTERFACE (INPUTS): .	104
Figure A-6:	STEPPER INTERFACE SPECIFICATION . .	105
Figure A-7:	STEPPER MOTOR INTERFACE (OUTPUTS) .	106
Figure A-8:	STEPPER INTERFACE SPECIFICATION . .	107
Figure A-9:	DC MOTOR INTERFACE MODULE	109
Figure A-10:	DC MOTOR INTERFACE SPECIFICATION .	110
Figure A-11:	ANALOG INPUT INTERFACE MODULE. . .	113
Figure A-12:	ANALOG INPUT INTERFACE SPECIFICATION	114
Figure A-13:	STEPPER MOTOR DRIVER MODULE A. . .	116
Figure A-14:	STEPPER MOTOR DRIVER MODULE B. . .	117
Figure A-15:	STEPPER DRIVE SPECIFICATION A. . .	118
Figure A-16:	STEPPER DRIVE SPECIFICATION B. . .	119
Figure A-17:	DC MOTOR DRIVER MODULE	121
Figure A-18:	DC MOTOR DRIVE SPECIFICATION . . .	122
Figure A-19:	SPEED SENSING MODULE	124
Figure A-20:	SPEED SENSING SPECIFICATION. . . .	125
Figure A-21:	INPUT CONDITIONING MODULE.	126
Figure A-22:	INPUT CONDITIONING SPECIFICATION .	127

List of Figures

	PAGE
Figure 1 : VIC-20 MEMORY MAP	8
Figure 2 : VIC-20 SYSTEM VARIABLES	9
Figure 3 : A LINE OF BASIC PROGRAM STORED IN VIC	15
Figure 4 : BASIC PROGRAM STORED IN MEMORY. . .	17
Figure 5 : STRUCTURE OF THE TINY BASIC COMPILER	19
Figure 6 : GRAMMAR OF THE TINY BASIC COMPILER-1	20
Figure 7 : GRAMMAR OF THE TINY BASIC COMPILER-2	21
Figure 8 : SEQUENCING FOR RECOGNITION OF AN EXPRESSION-1.	23
Figure 9 : SYSTEM ARCHITECTURE	26
Figure 10 : SYSTEM STRUCTURE.	29
Figure 12 : BIT ASSIGNMENT FOR CONTROL LANGUAGE	40
Figure 13 : STATE MACHINE OF THE CONTROL LANGUAGE.	44
Figure 14 : SEQUENCING FOR RECOGNITION OF AN EXPRESSION.	60
Figure A-1: I/O BUS EXPANSION MODULE.	98
Figure A-2: I/O BUS SPECIFICATIONS.	99

SCOPE OF THE THESIS

The object of this thesis was to collectively create a BASIC-like compiler with an extension toward I/O control in a real time control application. With additional primitives to handle I/O manipulation, A/D conversion, D/A control and a real time clock, modularization of a simple real time control system at low cost is achieved.

With the same minimum hardware possible, a VIC-20 and its accessories were chosen as a sample system to develop and to simulate the simple real time control system. The VIC-20 and its accessories are :

1. VIC-20
2. Disk drive or cassette tape
3. Printer

Besides the VIC-20 and its accessories, additional I/O of the system must be included with specifications as described below:

1. D.C. motor driver (some standard modules are

available , see interface specification for information .
Suggestion of a simple model and control circuit for
experimenting in this thesis are listed in Appendix A)

2. Switches, relays (standard parts used according to
manufacture specification. Single pole single throw DIP
switches and LED are used in demonstrating the capability of
the I/O compiler in this thesis.)

3. Stepper motor driver (standard modules are
available , see interface specification for detail .
Suggestion of simple model and control circuit for
experimenting in this thesis is listed in Appendix A)

Since the suggested control circuits and their
specifications are not part of the thesis the author would
recommend the user to seek technical advice among those who
are familiar with the subjects for implementing certain
applications with the use of this system.

In addition to software which is based on the Tiny
Basic Compiler, originally written by Mark Zimmerman and

David Malmberg and available from Abacus Software Company, P. O. Box 7211, Grand Rapids, MI 49510 , the control language was also developed.

As it was written, the restrictions of the Tiny Basic Compiler are :

1. Number of Lines of BASIC code : must be between 0 to 255.

2. Names of variables : only single letter variable names in programs which it compiles implying, that the variable's names are the 26 letter of the alphabet A through Z. More than 26 variables can not be accomodated as it was written.

3. IF statement : only legal IF statement is one which tests for a variable equal to zero : If zero, the statement is false and not executed, but if nonzero, program control is GOTO or GOSUB to the specified line number.

4. Arithmetic operation : there is valid only a single

operation (at most) on the right hand side of the equal sign of the equations which it compiles. Besides that, no numbers are allowed to appear in equations to be evaluated.

This thesis describes a modified version of the Tiny basic compiler with the following modification:

1. Number of Lines of BASIC code : 0 to 1000. This restriction can be removed with some modification if more lines are needed.

2. Names of Variables : could be single letter, letter and letter or letter and digit. Altogether, there are 962 ($= 26 + (26 \times 36)$) allowable variables.

1.0 INTRODUCTION

1.1 BACKGROUND

One significant difference between a personal computer and a minicomputer is that the lack of storage capacity and the lack of compiling capacity make the personal computer far from able to perform a complex program without difficulty.

One significant difference between a personal computer and a programmable controller is that the lack of dedicated I/O routines to perform operations on some special attached peripherals such as D.C. motors, stepper motors or other binary I/O operations.

The desire to increase execution speed and to perform I/O tasks while maintaining the generality of a personal computer brought forth the idea of putting together in one package, an I/O BASIC compiler for the VIC-20. The VIC-20 was chosen for its general purpose , low cost and ease of expansion. The VIC-20 and its peripherals are evaluated for

their potential for computer control implementation in the next section.

1.2 VIC-20 AND ACCESSORIES

The following are necessary parts of the overall project :

1. VIC-20 as controller.
2. Disk drive for floppy diskettes or data cassette recorder for cassette tape for storage.
3. Printer for listing of the compiled program.
4. Controlled objects.

1.3 BASIC INTERPRETER OF VIC-20

The investigation of VIC-20 as a way toward the scope of a real time system controller requires a full understanding of its execution, its storage and its I/O capacity. In the following sections we will study these elements in an comprehensive manner.

1.3.1 VIC USER MEMORY

The VIC-20 memory map in Figure 1 gives the general idea about its maximum capacity as a stand alone controller with dedicated I/O and its expansion.

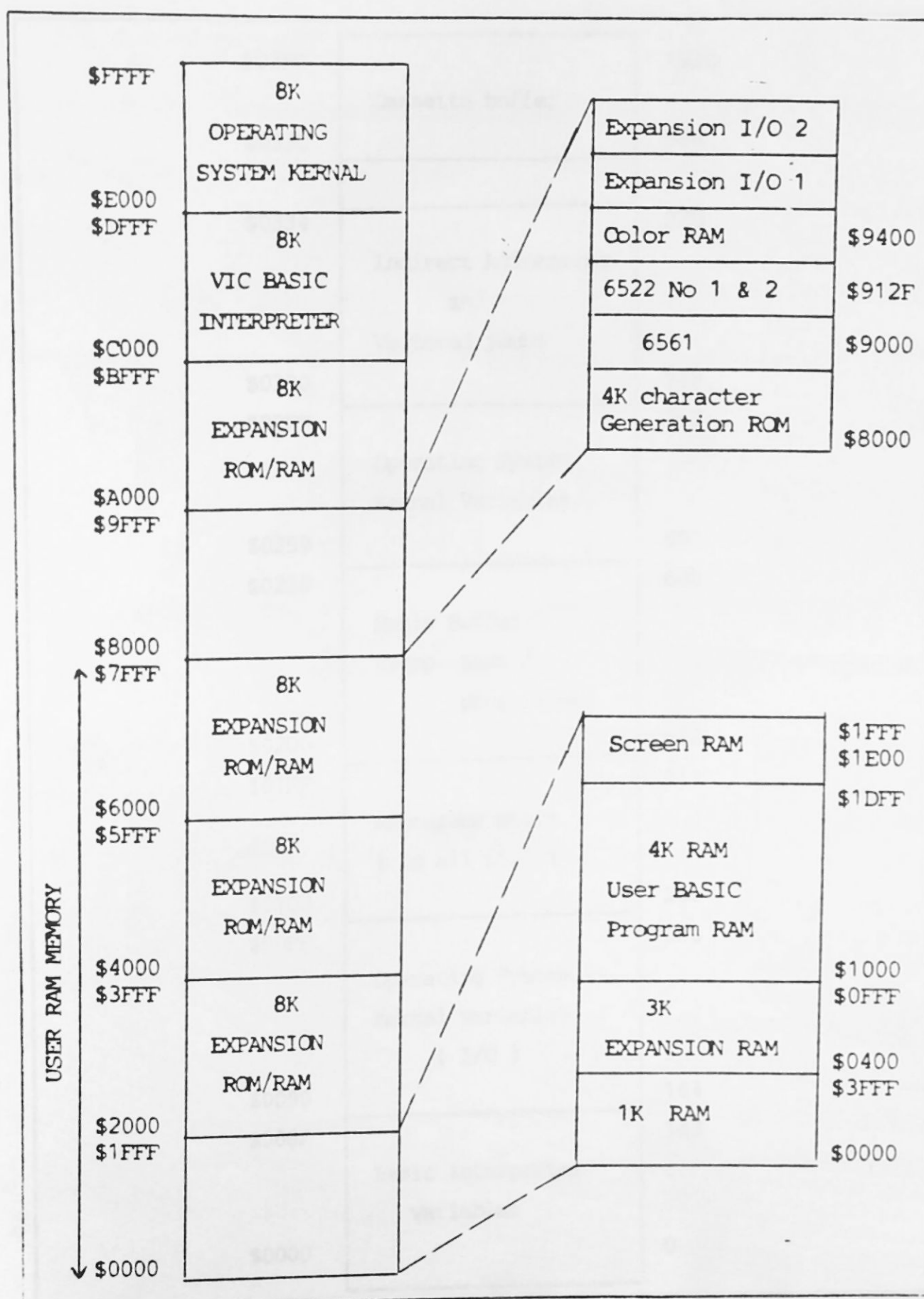


Figure 1 : VIC-20 MEMORY MAP

\$03FC	Cassette buffer	1020
\$033C		828
\$0334	Indirect Addressing and Vectored jumps	820
\$0300		768
\$02FF	Operating System Kernal Variables	767
\$0259		601
\$0258	Basic Buffer (Temp. text / prog. line)	600
\$0200		512
\$01FF	Processor stack (to all 6502)	511
\$0100		256
\$00FF	Operating System Kernal variables (I/O)	255
\$0090		144
\$008F	Basic Interpreter variables	143
\$0000		0

Figure 2: VIC-20 SYSTEM VARIABLES

1.3.2 I/O OPERATION

VIC-20 communicates with the user with peripheral devices via three integrated circuits :

1. 6561 VIC chip : This chip controls the video displays, sound generation, the peripheral devices, a light pen and a joystick.

2. Two 6522 VIA : They are used to perform all other I/O functions of VIC. The functioning of these chips is controlled by internal programmable registers. There are sixteen registers in each of the chips. These 32 registers are located in addressable memory space and are located at \$9110-\$912E. Thus, they can be accessed from Basic using PEEK and POKE statements and from machine code using LDA and STA statements.

1.3.3 SUBROUTINE CALLING

We further investigate the basic scheme of the VIC-20 operation to determine its capacity to perform I/O operations.

First, the VIC Basic Interpreter translates a high level basic program, step by step into a series of machine code routines. These perform the required function for each Basic command. Thus, utilizing these machine code routines to compile a basic language program into machine code is possible with much less effort than creating a full comprehensible one.

Second, the VIC has an advantage over many other small micro computer systems in that it can be programmed in both Basic and machine code. There are two ways to use commands in Basic to call a machine code subroutine: USR and SYS. The most powerful and flexible is SYS. Variables can be transferred between a Basic program and a machine code program by using PEEK and POKE. The only requirement with a SYS subroutine is that the last instruction in the subroutine is a RTS. This automatically returns control to the Basic program. This makes it possible to develop an I/O basic compiler with the interpreter without an expensive development system. I/O machine code routines can be developed to enhance the language and therefore achieve both goals in a simple process.

1.3.4 SPEED

The speed of execution of the VIC-20 is the most important factor in the proposed system since it does not offer adequate speed for the most part of the intended peripherals.

First, the execution speed of the VIC-20 is wasted by the line by line and function by function translation. The actual execution of the machine codes only takes a fraction of the total time .

Second, the machine code program and subprogram can be entered and executed via POKE and SYS but this also encounters the difficulty of full knowledge of the machine operation, and full knowledge of the machine language as well.

Third, in both previous cases the program can not be flexible enough to be verified and enhanced without difficulty to the user. For the user to run a high level language program at machine speed to execute I/O operations and with the flexibility to verify or modify the program in

one step, the I/O basic compiler thus encompasses the original Basic interpreter in both speed and convenience.

Furthermore, the implementation of I/O dedicated instructions allow the user to expand the full capacity of the VIC-20 to manipulate the peripherals in multiple ways at a high level language.

2.0 TINY BASIC COMPILER

The selection of software to implement the proposed system is based on the Tiny Basic Compiler for a simple reason: the availability of this compiler and its condensed form make it easy to implement a programmable controller-like language . Let us see what the Tiny Basic Compiler in its own form would be like in order to convert the source to machine code program.

2.1 EDITING TEXT --- VIC-20 ---

First, the VIC-20 basic interpreter is used to edit a program: The Basic program is stored from location 4097 decimal upwards (if the 3K expansion is fitted then programs start at location 1025 decimal) and strings and variables are stored from top of memory downwards.

link address	line number	text	end of line
--------------	-------------	------	-------------

When a program line is entered on the keyboard, it is first written into the keyboard buffer. The operating system then transfers it byte by byte as it is entered onto the

screen. The line however is not entered into memory until a carriage return is pressed. This causes the operating system to transfer the program line just entered from the screen into memory via the Basic buffer where the line of code is compressed and formatted. Each line is stored in a specific format using a compressed version of the Basic text. This reduces the memory requirements of a program and allows longer programs to be run. An useful result of text compression is an shorthand way of writing Basic commands either in the program or direct command mode. This relies on the fact that the routine which converts commands to tokens looks only at the first two characters of a command word.

A line of a program is entered on VIC by the following format:

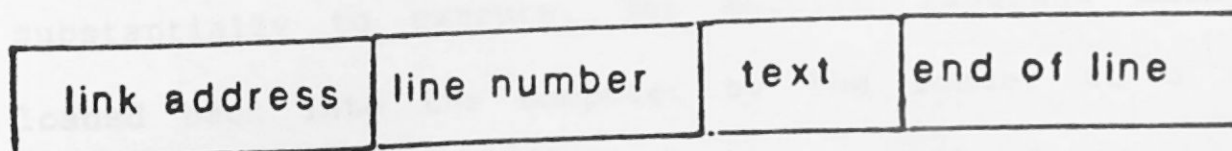


Figure 3 : A LINE OF BASIC PROGRAM STORED IN VIC

2.2 HOW DOES THE TINY BASIC COMPILER WORK ?

The Tiny Basic Compiler is a translator for floating point arithmetic operations. It is a Basic program that scans through an user Basic program and converts these functions into machine code programs which in turn will be executed later. Then it writes out the sequence of machine language onto cassette tape drive or a disk. When finished, a machine language program will be ready for execution. The compiling process is designed so that it can be used separately from the execution process to reserve the memory usage both by the compiler and the user program. The machine code program once generated will be written directly onto the tape or disk which also reduce the memory usage substantially to execute. The machine language must be loaded back into the computer by the loader to a user specified memory location thus making it possible to run the program from relative memory space.

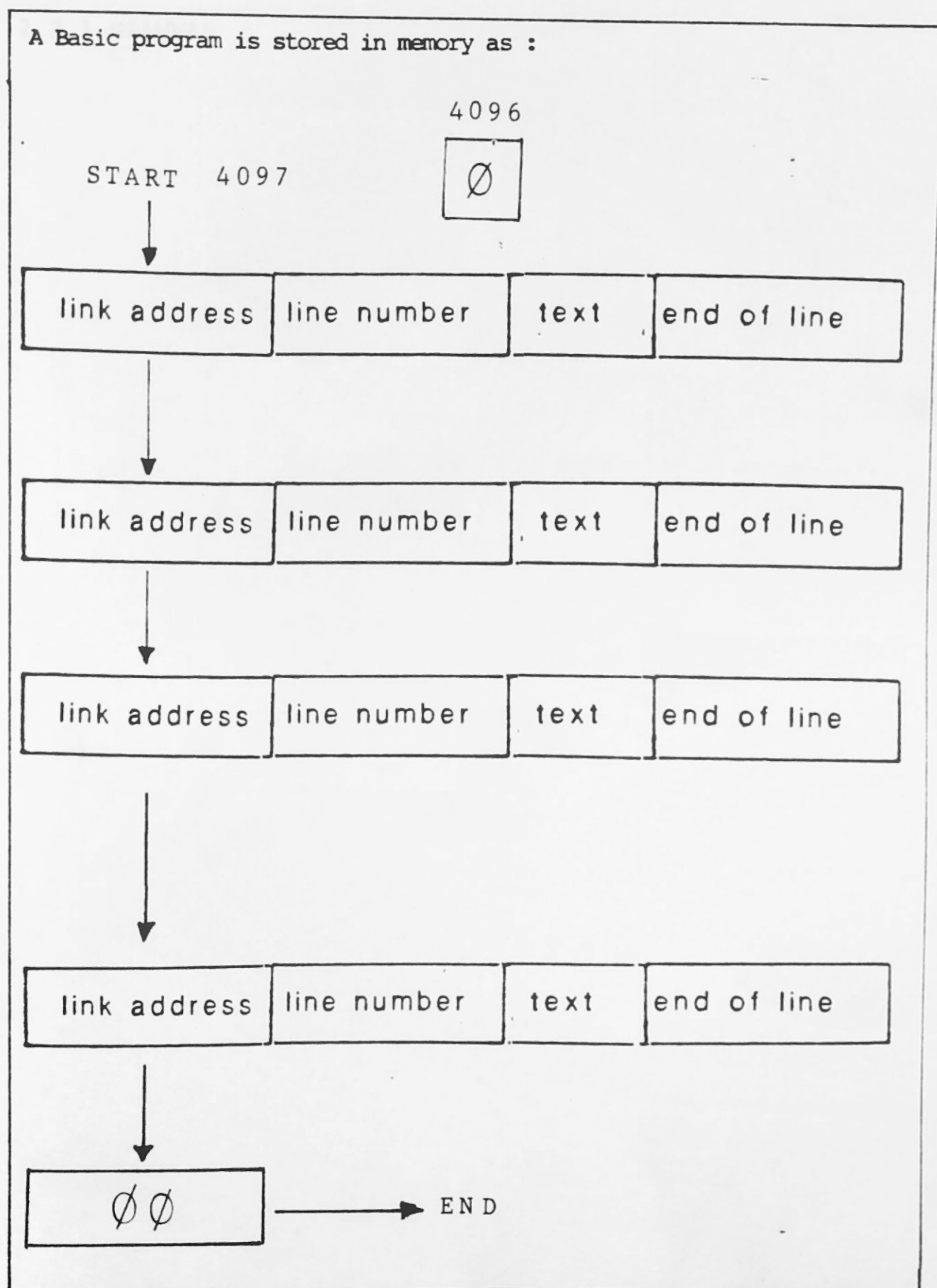


Figure 4 : BASIC PROGRAM STORED IN MEMORY

2.2.1 GRAMMAR



Figure 5.1 STRUCTURE OF THE TINY BASIC COMPILER

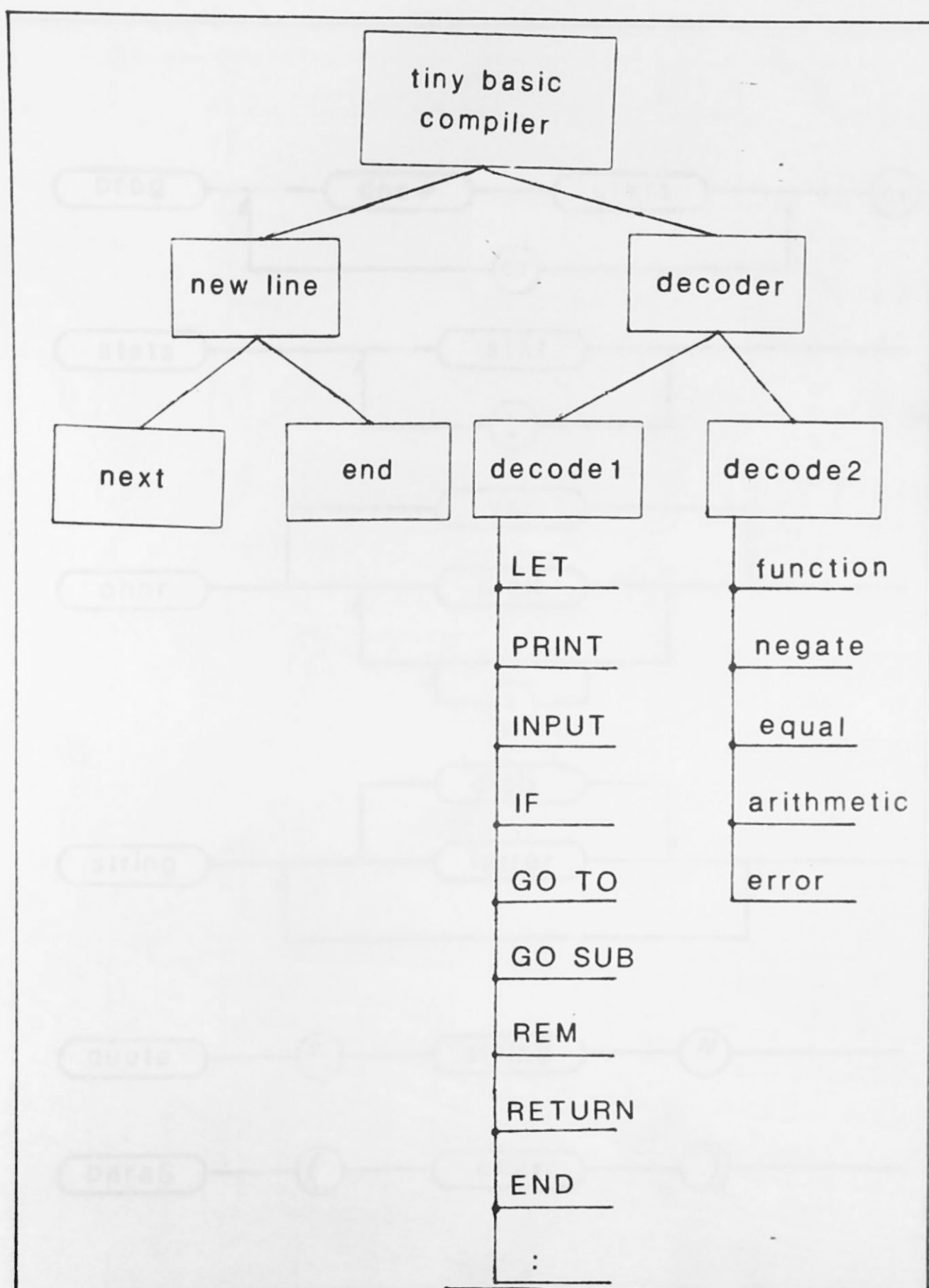


Figure 5 : STRUCTURE OF THE TINY BASIC COMPILER

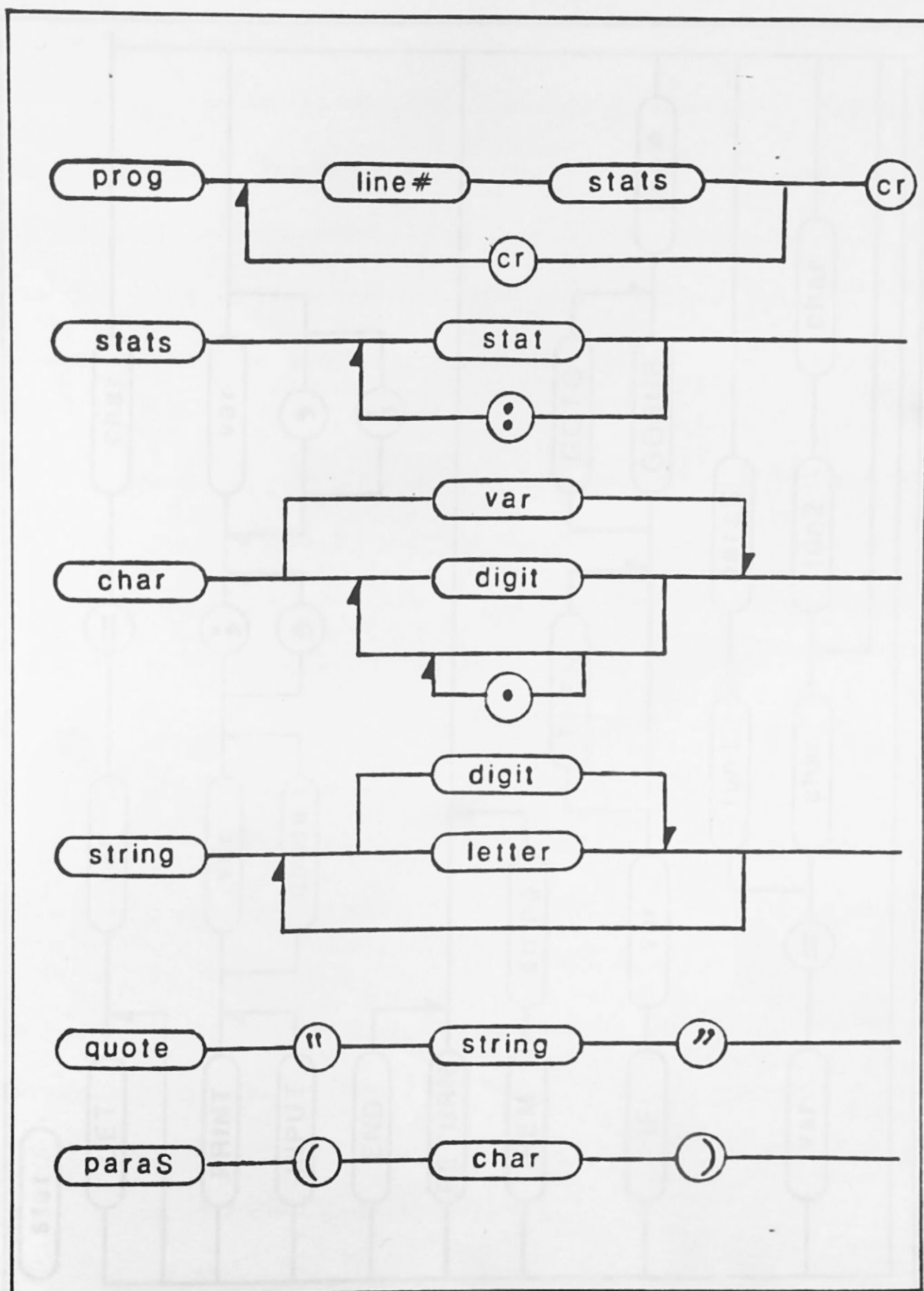


Figure 6 : GRAMMAR OF THE TINY BASIC COMPILER-1

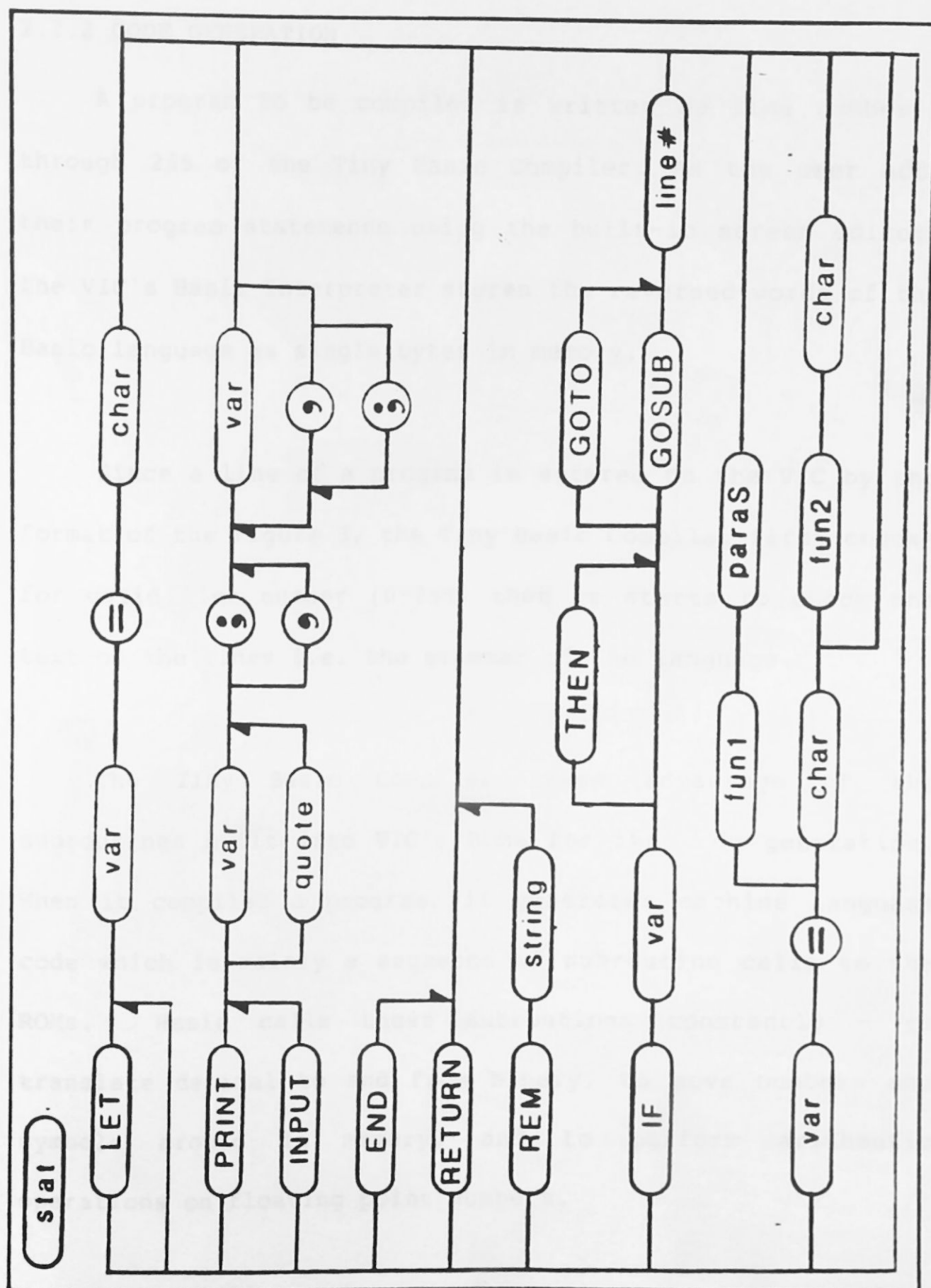


Figure 7 : GRAMMAR OF THE TINY BASIC COMPILER-2

2.2.2 CODE GENERATION

A program to be compiled is written as line number 0 through 255 of the Tiny Basic Compiler. As the user adds their program statements using the built-in screen editor, the VIC's Basic Interpreter stores the reversed words of the Basic language as single bytes in memory.

Since a line of a program is entered on the VIC by the format of the Figure 3, the Tiny Basic Compiler first checks for valid line number (0-255) then it starts to check the text of the lines i.e. the grammar of the language.

The Tiny Basic Compiler takes advantage of the subroutines built into VIC's ROMs for its code generation. When it compiles a program, it generates machine language code which is mainly a sequence of subroutine calls to the ROMs. Basic calls these subroutines constantly - to translate decimal to and from binary, to move numbers and symbols around in memory, and to perform arithmetic operations on floating point numbers.

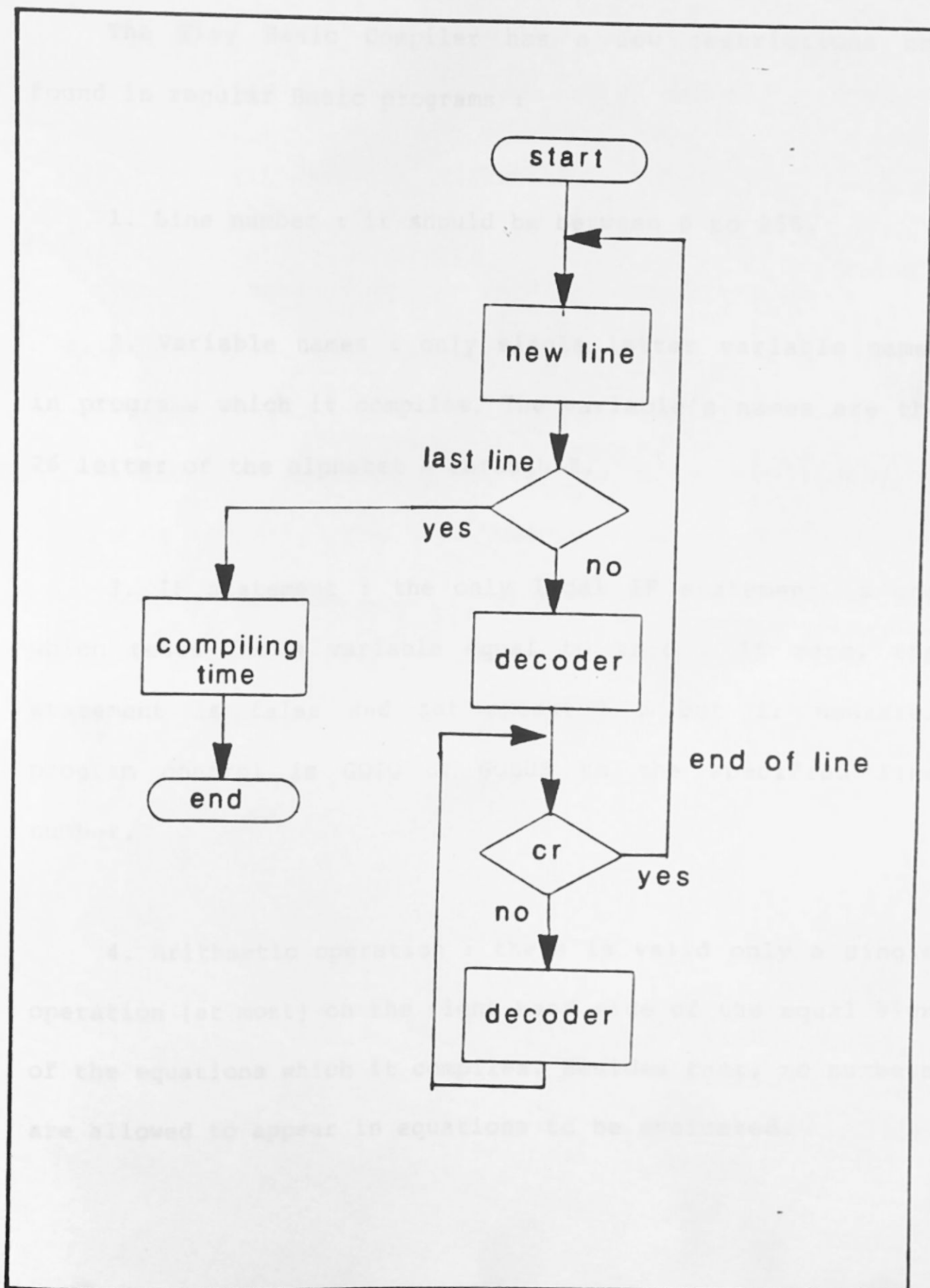


Figure 8 : SEQUENCING FOR RECOGNITION OF AN EXPRESSION-1

The Tiny Basic Compiler has a few restrictions not found in regular Basic programs :

1. Line number : it should be between 0 to 255.
2. Variable names : only single letter variable names in programs which it compiles. The variable's names are the 26 letter of the alphabet A through Z.
3. IF statement : the only legal IF statement is one which tests for a variable equal to zero : If zero, the statement is false and not executed , but if nonzero, program control is GOTO or GOSUB to the specified line number.
4. Arithmetic operation : there is valid only a single operation (at most) on the right hand side of the equal sign of the equations which it compiles. Besides that, no numbers are allowed to appear in equations to be evaluated.

3.0 SYSTEM OVERVIEW

3.1 HARDWARE (SYSTEM REQUIREMENTS)

The I/O Basic Compiler can be visualized as an I/O dedicated package which references its I/O with a logical name . The logical name in turn associated with a physical I/O device (it could be a D.C. motor, a stepping motor, a switch or relay). The user must pay special attention to the hardware configuration of the system as a whole for connecting or installing such peripherals. From the programming standpoint the logical name will not interfere or impair the user from executing his program effectively.

Figure 3 : SYSTEM ARCHITECTURE

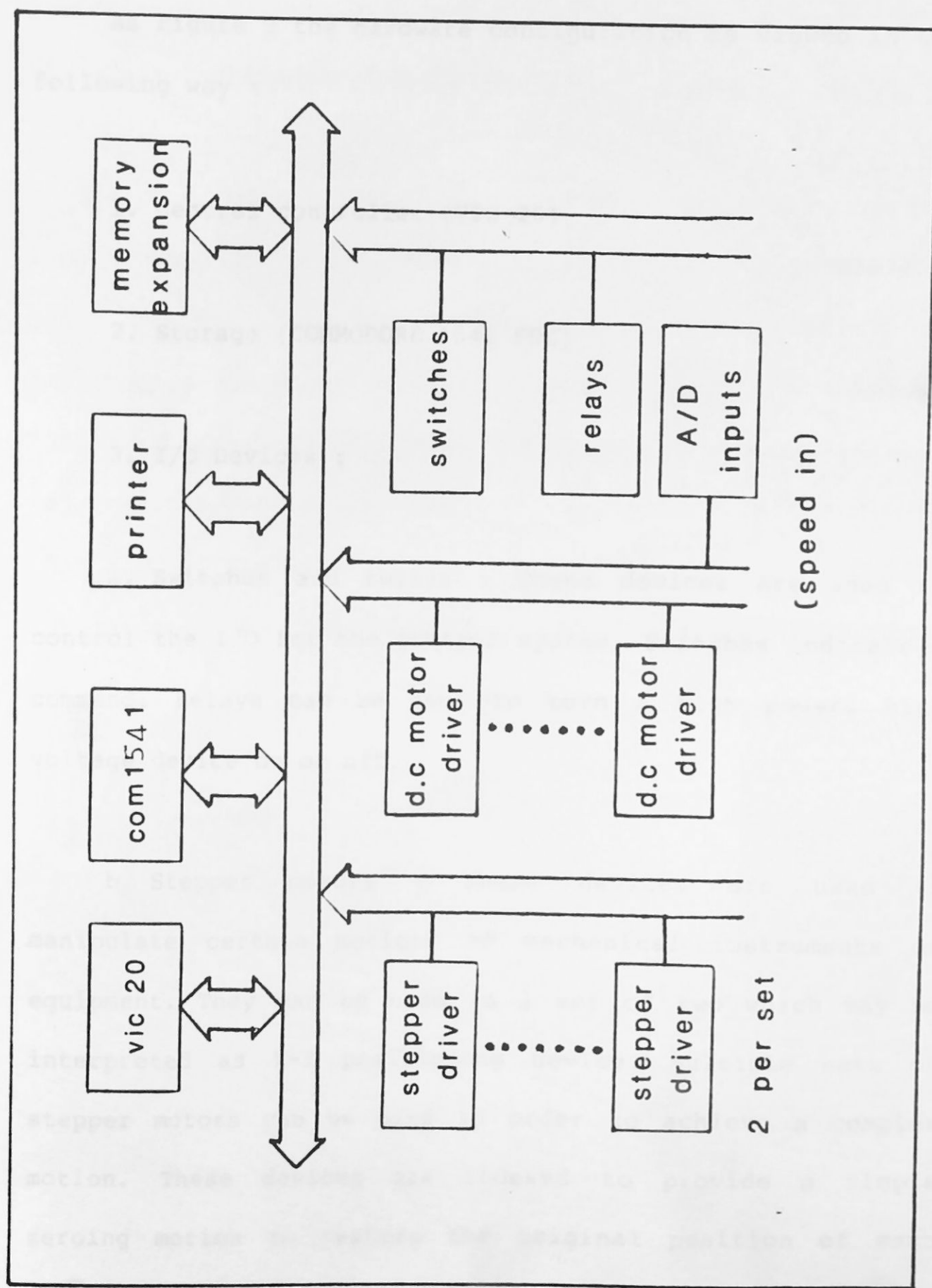


Figure 9 : SYSTEM ARCHITECTURE

As Figure 9 the hardware configuration is viewed in the following way :

1. Central controller (VIC-20)

2. Storage (COMMODORE 1541 FDC)

3. I/O Devices :

- a. Switches and relays : These devices are used to control the I/O for the control system. Switches indicate a command. Relays can be used to turn a high power, high voltage device on or off.

- b. Stepper motors : These devices are used to manipulate certain motions of mechanical instruments or equipment. They can be used in a set of two which may be interpreted as X-Y positioning device. Multiple sets of stepper motors can be used in order to achieve a complex motion. These devices are indexed to provide a simple zeroing motion to restore the original position of each motor, a single set, or all the motors in the system.

Direction is also used to simplify the quick implementation of shortest distance of travel.

c. D.C. motors : This could be an open loop or closed loop control. The interface with VIC-20 can be simple via D/A (digital to analog) and A/D (analog to digital) to implement a speed control with a resolution of 128 steps or 128 speed levels. The actual speed can be set to certain requirements of the user and this is flexible as the driver that handles the speed/torque requirement. It also controls the motor ON/OFF condition for programming convenience.



Figure 10 : SYSTEM STRUCTURE

As Figure 10 the system configuration is viewed as the

following :

3.2 SOFTWARE (SYSTEM STRUCTURE)

This compiler is derived directly from the Tiny Basic compiler. The authors are Mark Zimmerman and David Halpern, from Abacus Software Company, P. O. Box 7111, Grand Rapids, MI 49510. In addition to the Tiny Basic Compiler which is a

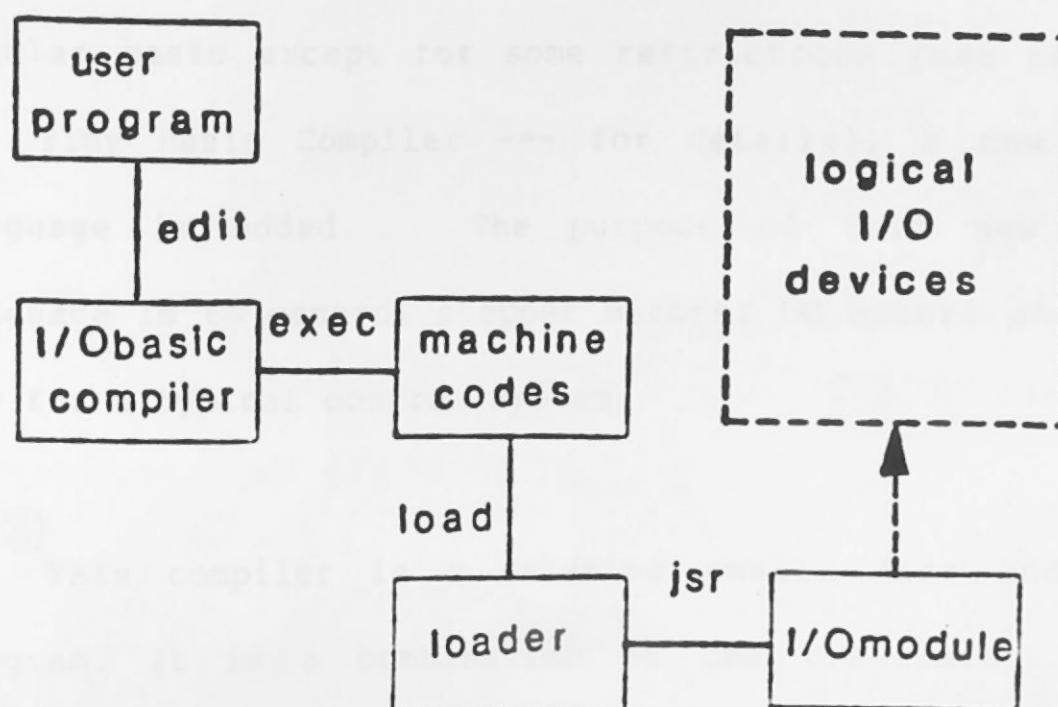


Figure 10 : SYSTEM STRUCTURE

As Figure 10 the system configuration is viewed as the following :

1. Tiny I/O Basic Compiler :

This compiler is derived directly from the Tiny Basic compiler. Its authors are Mark Zimmerman and David Malmberg, from Abacus Software Company, P. O. Box 7211, Grand Rapids, MI 49510. In addition to the Tiny Basic Compiler which is a regular basic except for some restrictions (see chapter 2 --- Tiny Basic Compiler --- for details), a new control language is added . The purpose of this new control language is to control stepper motors, DC motors and binary I/O for a typical control system.

This compiler is a relative small, fast and simple program. It is a combination of the translator and the assembler. The translator interprets the symbols in a line. At the same time, when the translator discovers a new symbol, the assembler generates machine language code which is mainly a sequence of subroutine calls to the ROMs. Those machine language codes are also outputted to a disk file for later execution.

It also has the option for the user to view the

assembly language program during the compiling process which is advantageous for debugging if there is an error.

Thus, the input for this compiler is the user's program and its output is a disk or cassette tape file which contains the no error machine language codes.

2. Loader :

The loader is a program which loads the object code file in the disk into memory area . It invokes the subroutines which are built into the ROMs .

Thus, its input is the object code file in the disk, and its output is the subroutine calls to the ROMs.

3. I/O modules :

These I/O modules are built to accommodate the I/O commands in conjunction with other existing I/O kernels in the VIC-20 operating system. The I/O modules also intercept the interrupt vector at the power on reset as well as warm

boot via RESTORE to provide multitasking capability. In normal operation, the VIC-20 uses the real-time clock interrupt to perform display refresh, scan the keyboard and update the time. These background operations provide a set of communication facilities to the foreground operation which in turn responds to key entries and perform the requested functions.

When a basic program is running, the foreground operation consists of interpreting the program line by line, then making system calls to the Basic kernel.

In intercepting the interrupt, the I/O modules add more functions which need to be done at regular intervals such as stepping of the motor, reading the speed channel and reading the analog channel so that the time between steps and the time between the channels are not wasted. The I/O modules also provide an image of each device in the process as well as the mechanism to perform the I/O operation. Since the I/O modules are resident in the operating system in the form of an EPROM, the compilation of the I/O commands will be a lot easier and less much less memory requirement.

3.2.1 I/O CONTROL SUBROUTINE ADDRESSES

These are the addresses of the subroutines which are used for the control language :

1. SETUP (\$A000)
2. INIT (\$A00A)
3. IALL (\$A00D)
4. MOVE (\$A010)
5. MSET (\$A013)
6. MALL (\$A016)
8. STRT (\$A019)
9. RALL (\$A01C)
10. STPM (\$A01F)
11. SALL (\$A022)
12. IBIT (\$A025)
13. OBIT (\$A028)
14. STIM (\$A02B)
15. RTIM (\$A02E)
16. SAMPLE(\$A031)
17. SPEED (\$A034)
18. CHAN (\$A037)

This is the link table that will accommodate the transaction of the compiler and the loader.

```

:      This is the link between the compiler to the i/o module
:      It is intended to serve the multitasking among BASIC i/o
:      kernal and the CONTROL i/o kernal
:      the multitasking of this module is performed via a set of
:      i/o queues named as IMAGES
:      the link table will provide a fast access to variables
:      insteads of the BASIC variable table
:      Each of set of similar i/o will be recorded sequentially
:      to modularize the software package.

```

5F00	B	=	85F00	
5F00	CTL8L	=	8	
5F01	DCMDN	=	8+1	:BIT IMAGE OF THE ON STATE
5F02	SPEED1	=	8+2	:READ FROM A/D PER INTERRUPT
5F03	SPEED2	=	8+3	:4 CHANNELS AT 1/60 SECONDS
5F04	SPEED3	=	8+4	:THUS THE CONVERSION RATE IS
5F05	SPEED4	=	8+5	:ABOUT 1/15 FOR ANY CHANNEL
5F06	SPEED5	=	8+6	:WHICH MEANS THAT THE CHANNEL
5F07	SPEED6	=	8+7	:IS READ AT THE RATE OF 15
5F08	SPEED7	=	8+8	:READS PER SECOND
5F09	SPEED8	=	8+9	:-----
5F0A	CHN1	=	8+10	:THE SAME THING APPLY TO
5F0B	CHN2	=	8+11	:THE ANALOG CHANNEL HERE
5F0C	CHN3	=	8+12	:THE VALUES WILL BE STORED
5F0D	CHN4	=	8+13	:AND UPDATED PER INTERRUPT
5F0E	CHN5	=	8+14	:THE COMMAND LANGUAGE,
5F0F	CHN6	=	8+15	:WILL BE DIRECTED TO READ
5F10	CHN7	=	8+16	:THE CORRESPONDING LOCATION
5F11	CHN8	=	8+17	:FROM THIS TABLE
5F12	DCF2F	=	8+18	:-----
5F13	STP15	=	8+19	:THE STEPS
5F14	STP25	=	8+20	:ARE REGISTERED HERE
5F15	STP35	=	8+21	:TO BE SERVICE IN
5F16	STP45	=	8+22	:EVERY INTERRUPT
5F17	STP55	=	8+23	:THIS WILL CUT DOWN THE
5F18	STP65	=	8+24	:WASTED TIME DURING
5F19	STP75	=	8+25	:MOTOR STEPPING
5F1A	STP85	=	8+26	:-----
5F2A	DCMSP1	=	8+43	:THE COMMAND START-UP
5F2C	DCMSP2	=	8+44	:WILL REGISTER THE SPEED HERE
5F2D	DCMSP3	=	8+45	:AS LONG AS THE PROGRAM
5F2E	DCMSP4	=	8+46	:ALLOW IT TO BE THERE
5F2F	DCMSP5	=	8+47	:STOP WILL NOT ERASE THESE
5F30	DCMSP6	=	8+48	:RECORDED VALUES
5F31	DCMSP7	=	8+49	:THE SPEED RANGE 10-107
5F32	DCMSP8	=	8+50	:-----

3.3 I/O MEMORY MAP FOR CONTROL LANGUAGE

This section contains the following parts :

1. I/O and VIA
2. Stepper motors and VIA
3. D.C. motors and VIA
4. The local control table for I/O routines
5. Necessary subroutines for passing floating accumulator

3.3.1 INPUT/OUTPUT AND VIA INTERFACE

The interface between the switches , the relays and VIA is as follows :

\$9800 : Input port

\$9802 : Input control port

\$9801 : Output port

\$9803 : Output control port

\$9902 : Stepper motor control port

\$9901 : Stepper motor direction control port

3.3.2 STEPPER MOTOR AND VIA INTERFACE

The interface between the stepper motors and VIA is as follows :

\$9900 : stepper motor select port

Each stepper motor is connected to VIC-20 by a bit of this output port. The stepper motor is stepped one step when the pulse is high and it is stopped when the pulse is low.

\$9901 : stepper motor output direction port

Each direction of a specified stepper motor is controlled by a bit in this port. If the content of a bit is 1 then the direction is counter clockwise. Otherwise, if it is 0 then the direction is clockwise.

\$9902 : Stepper motor control port

\$9903 : Stepper motor direction control port

\$9A00 : Index port

Each bit in this port is used to control the index of the stepper motor. The stepper motor is at theta zero when the signal of this index is low.

\$9A02 : Index control port

\$9A01 : Return port

Each bit in this port is used to indicate the end of the stepping rate of the stepper motor via a one shot circuit connected to this port. The stepper motor can be stepped at the rate of multiples of 1/60 of a second with respect to where the length of the one shot ends between two intervals.

\$9A03 : Return control

3.3.3 DC MOTORS AND VIA INTERFACE

\$9B00 : DC motor select port

\$9B02 : DC motor select control

\$9B01 : DC motor speed port

\$9B03 : DC motor speed control

There are eight bits to be used to control a DC motor.

Bit 0 : turn DC motor on (0) or off (1)

Bit 1-7 : to be connected to D-A, which are used to control the speed of DC motors. Thus there are 128 different speeds at which the D.C. motors could be controlled.

\$9C00 : Channel select port

\$9C01 : A/D converter input port

\$9C02 : Channel select control

\$9C03 : A/D converted input control

This is the data acquisition facility of the system. The closed-loop control can be implemented with the real time facility. The timing required by the A/D conversion may not be a restriction if sample time is not less than 1/10 of a second. The A/D conversion is performed via Interrupt

Service routine to put the input data in an array. Thus, the commands SPEED# and CHANNEL# are reading the value in memory location instead.

The mechanism in A/D conversion is mainly a sequence of:

1. read the last channel and store it in array;
2. write the new channel select and start the conversion;

The sequence is then repeated for all 16 channels. The bandwidth of the system is determined from this algorithm at a maximum of 20 HZ. D Slower sampling rate can be obtained by the SAMPLE# command. Sampling time from 1 to 15 will gives the bandwidth from 20 HZ down to 7 HZ.

The 8-bits resolution input is considered a standard for 8-bit microprocessor. The 16 channels will be plenty for any application.

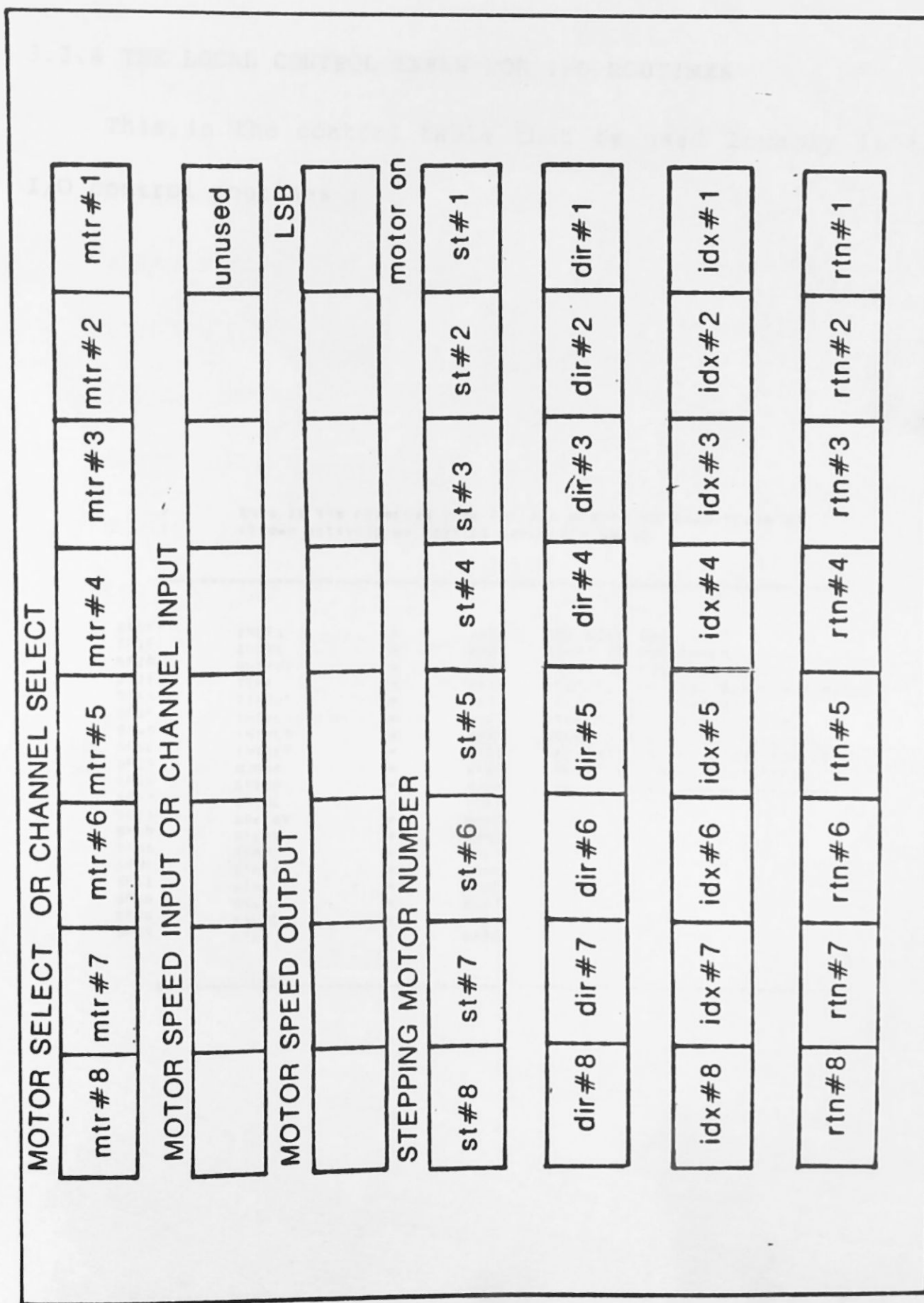


Figure 11: BIT ASSIGNMENT FOR CONTROL LANGUAGE

3.3.4 THE LOCAL CONTROL TABLE FOR I/O ROUTINES

This is the control table that is used locally in the I/O control routines :

SDR12 : MIF

SDR17 : FCM

SF06 : STIME

SF0E : STIME

;
: this is the reserved area for i/o module to keep track of
: its own activity as well as temporary usage

```

-----
SF1E      INDEX      =      B+30      :BE USED EACH STEP
SF1F      INBYT      =      B+31      :WHAT IS OUT THERE
SF20      GUTRYT     =      B+32      :WHAT IS GOING OUT NOW
SF21      TIME       =      B+33      :RECORD THE TIME BETWEEN TIMEOUT
SF4A      TIMEOUT    =      B+74      :RECORD THE TIMEOUT
SF3F      INTVL      =      B+63      :RECORD THE SAMPLING TIME
SF4D      INTVL0     =      B+77      :RECORD THE ACTUAL FUNCTION
SF4C      INTREG     =      B+76      :AS THIS IS THE COUNT FOR IT
SF25      NUMBR      =      B+37      :TEMPORARY WORK REGISTERS
SF26      SETSP      =      B+38      :-----
SF27      SETDC      =      B+39      :
SF28      SDCLAY     =      B+40      :
SF29      DELAY      =      B+41      :
SF33      TAMP       =      B+51      :
SF3E      STPEA      =      B+62      :
SF41      @IN        =      B+65      :
SF43      RSLT       =      B+67      :
SF24      RSLTDV     =      B+36      :
SF49      RFEED      =      B+72      :
-----

```

SF00 : -----

SDR17 : INTIN

3.3.5 NECESSARY SUBROUTINES FOR PASSING FLOATING ACCUMULATOR

\$D391 : I2F

\$D1AA : F2I

\$DBA2 : M2F

\$DBD7 : F2M

\$FFDB : STIME

\$FFDE : RTIME

\$C000 : BASIC

\$FFE1 : STOP

\$F734 : UDTIM

\$FDB8 : RAMTAS

\$FD52 : MOVOSI

\$FDF9 : IOINIT

\$E518 : CINT

\$FF56 : PREND

\$E378 : INITNV

4.0 SYSTEM ARCHITECTURE AND CAPACITY

This chapter contains the following sections :

1. Control language
2. Capacity
3. Loader's specification
4. Loading and execution

4.1 CONTROL LANGUAGE

This section contains the following parts :

- a. Grammar of the control language
- b. Synthesis

4.1.1 GRAMMAR OF THE CONTROL LANGUAGE

The following figure is the state machine of the control language :

Figure 12 : STATE MACHINE OF THE CONTROL LANGUAGE

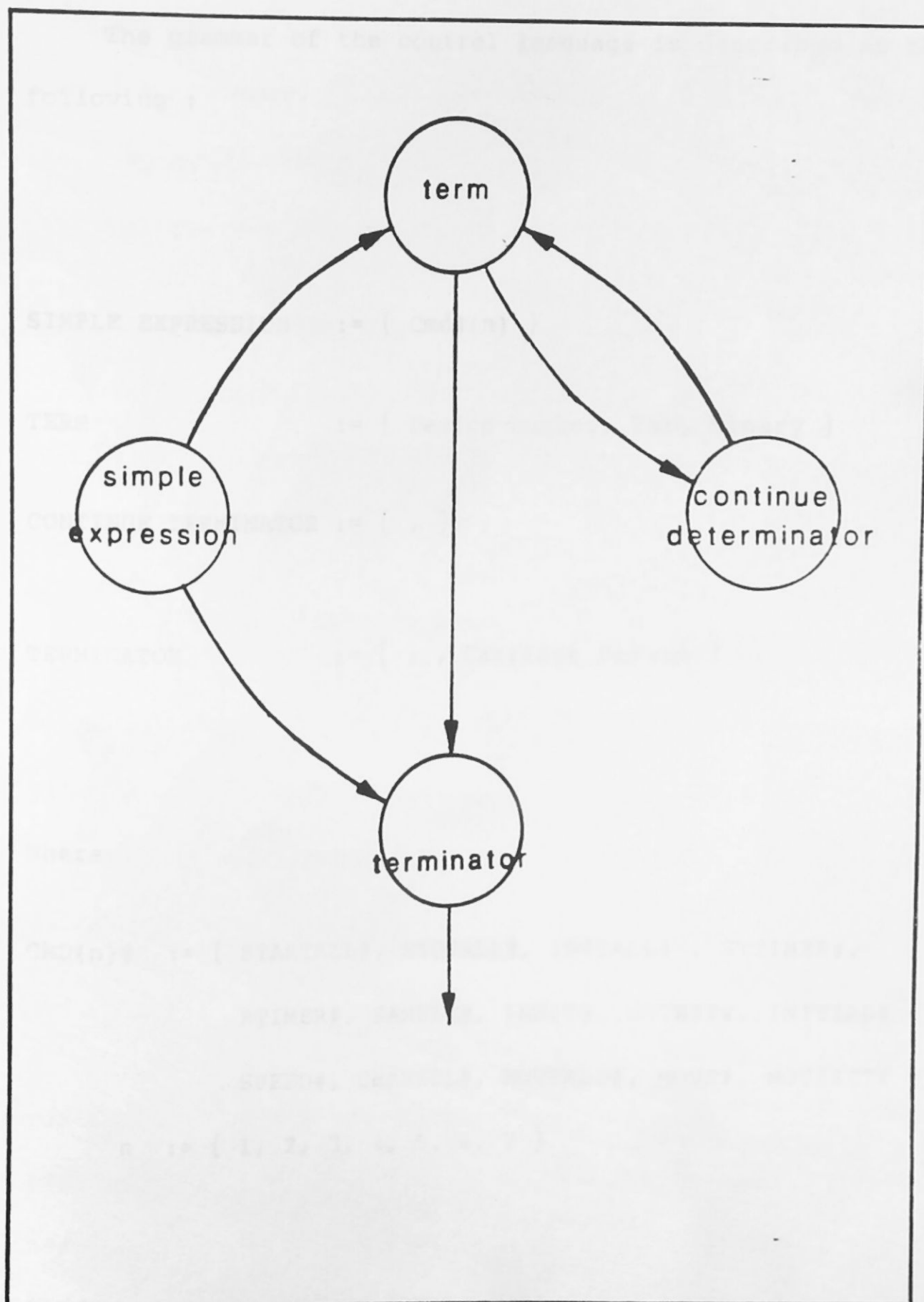


Figure 12: STATE MACHINE OF THE CONTROL LANGUAGE

The grammar of the control language is described as the following :

SIMPLE EXPRESSION := { Cmd#(n) }

TERM := { Device-number, Var, Binary }

CONTINUE TERMINATOR := { , }

TERMINATOR := { : , Carriage Return }

Where :

CMD(n)# := { STARTALL#, STQPALL#, INITALL# , STTIMER#,
RTIMER#, SAMPLE#, INBIT#, OUTBIT#, INITIAL#
SPEED#, CHANNEL#, MOVEALL#, MOVE#, MOVESET# }

n := { 1, 2, 3, 4, 5, 6, 7 }

DEVICE-NUMBER :=

:= { Bit-number,

DC-motor-number, Stepper-motor-number,

Stepper-motor-set-number, Channel-number }

VAR := { I/O, DIRECTION, STEP

SPEED, VALUE, TIME, INTERVAL }

BINARY := { DIRECTION }

Where :

BIT-NUMBER := { 0, 1, 2, 3, 4, 5, 6, 7 }

DC-MOTOR-NUMBER := { 1, 2, 3, 4, 5, 6, 7, 8 }

STEPPER-MOTOR-NUMBER := { 1, 2, 3, 4, 5, 6, 7, 8 }

STEPPER-MOTOR-SET-NUM := { 1, 2, 3, 4 }

CHANNEL-NUMBER := { 1, 2, 3, 4, 5, 6, 7, 8 }

I/O := { ON, OFF } = { 0, 1 }

DIRECTION := { Clock, Counterclock } = { 0, 1 }

STEP := { 1, 2, 200 }

SPEED := { 0, 1, 127 }

VALUE := { 0, 1, 127 }

TIME := { 0, 1, 9999 }

INTERVAL := { 0, 1, 15 }

1. CMD1: DEVICE-NUMBER

STOP: DC-MOTOR-NUMBER

2. CMD2: VAR:

START: TIME

FINISH: TIME

DELTA: INTERVAL

3. CMD3: DEVICE-NUMBER, VAR

END: BIT-NUMBER, 1/0

CUT: BIT-NUMBER, 1/0

SPEED: DC-MOTOR-NUMBER, SPEED

CHANNEL: CHANNEL-NUMBER, VALUE

4. CMD4: DEVICE-NUMBER, BINARY

INITIAL: STEPPER-MOTOR-NUMBER, DIRECTION

5. CMD5: VAR, BINARY

MOVE: STEP, DIRECTION

6. CMD6: DEVICE-NUMBER, VAR, BINARY

SUMMARY OF THE CONTROL LANGUAGE

1. CMD1# :

STARTALL#

STQPALL#

INITALL#

2. CMD2# DEVICE-NUMBER :

STQP# DC-MOTOR-NUMBER

3. CMD3# VAR:

STTIMER# TIME

RTIMER# TIME

SAMPLE# INTERVAL

4. CMD5# DEVICE-NUMBER, VAR :

INBIT# BIT-NUMBER, I/O

OUTBIT# BIT-NUMBER, I/O

SPEED# DC-MOTOR-NUMBER, SPEED

CHANNEL# CHANNEL-NUMBER, VALUE

5. CMD6# DEVICE-NUMBER, BINARY :

INITIAL# STEPPER-MOTOR-NUMBER, DIRECTION

6. CMD7# VAR, BINARY :

MOVEALL# STEP, DIRECTION

7. CMD8# DEVICE-NUMBER, VAR, BINARY :

MOVE# STEPPER-MOTOR-NUMBER, STEP, DIRECTION

MOVESET# STEPPER-MOTOR-SET-NUM, STEP, DIRECTION

1. INWRITE BIT-NUMBER, VAR

Input : BIT-NUMBER

Output : VAR = status of a specified bit

2. OUTWRITE BIT-NUMBER, VAR

Input : BIT-NUMBER

VAR (value)

Output : VAR = a specified bit is set/cleared according
to the entry value of 'VAR'

3. INITIALS STEPPER-MOTOR-NUMBER, DIRECTION

Input : STEPPER-MOTOR-NUMBER

DIRECTION

Output : Stepper motor number n is at 'start' zero

where n is an integer between 1..8

4. INITALLS

Input : None

Output : All stepper motors are at 'start' zero

5. MOVES STEPPER-MOTOR-NUMBER, STEP, DIRECTION

Input : STEPPER-MOTOR-NUMBER

DETAIL OF THE CONTROL LANGUAGE

1. INBIT# BIT-NUMBER, VAR

Input : BIT-NUMBER

Output : VAR = status of a specified bit

2. OUTBIT# BIT-NUMBER, VAR

Input : BIT-NUMBER

VAR (name)

output : VAR = a specified bit is on/off , according
to the entry value of 'VAR'.

3. INITIAL# STEPPER-MOTOR-NUMBER, DIRECTION

Input : STEPPER-MOTOR-NUMBER

DIRECTION

Output : Stepper motor number n is at theta zero
where n is an integer between[1,8] .

4. INITALL#

Input : None

Output : All stepper motors are at theta zero.

5. MOVE# STEPPER-MOTOR-NUMBER, STEP, DIRECTION

Input : STEPPER-MOTOR-NUMBER

5. STARTALL# STEP
DIRECTION
Output : A specified stepper motor is stepping ...
in direction ...
6. MOVESET# STEPPER-MOTOR-SET-NUM, STEP, DIRECTION
Input : STEPPER-MOTOR-SET-NUMBER
STEP
DIRECTION
Output : Stepper motor set number n is stepping ...
in direction ...
where n is an integer between [1,8]
7. MOVEALL# STEP, DIRECTION
Input : STEP
DIRECTION
output : All stepper motors are stepping ...
in direction ...
8. START# DC-MOTOR-NUMBER, SPEED
Input : DC-MOTOR-NUMBER
SPEED
Output : A specified DC motor is running at speed ...

9. STARTALL# :

Input : none

Output : All DC motors are running

10. STQP# DC_MOTOR_NUMBER

Input : DC-MOTOR-NUMBER

Output : A specified DC motor is stopped.

11. STQPALL#

Input : None

Output : All DC motors are stopped.

12. STTIMER# VAR

Input : VAR (name)

Output : The real time clock will be initialized at zero value, continues to count up to value in the variable location and also in a reserved location named TIMOUT

13. RTIMER# VAR

Input : VAR (name)

Output : VAR =

Value of time which is contained in reserved location named TIME. If TIMOUT is reached ,

the real time clock will be restarted and

TIME will be equal to zero.

(each clock = 1/60 seconds) .

14. SAMPLE# VAR

Input : VAR = positive integer

The value of VAR will be stored in a reserved
location named INTERVAL used to sample the
analog input .

(each clock = 1/20 seconds.)

Output : none

15. SPEED# DC-MOTOR-NUMBER, VAR

Input : DC-MOTOR-NUMBER

VAR (name)

Output : VAR =

= Speed of a specified DC motor via frequency
to voltage converter and A/D converter.

16. CHANNEL# CHANNEL-NUMBER, VAR

Input : CHANNEL-NUMBER

VAR (name)

Output : VAR =

= Analog input from a specified physical
channel of A/D.

4.1.2 SYNTHESIS

There are three main parts in this section as follows :

1. Word identification,
2. Code generation,
3. Detailed description of the control language.

4.1.2.1 WORD IDENTIFICATION

From the restriction of the Tiny Basic Compiler, the variable's name should be defined as only one letter. To have more variables available, in this thesis, a new construction of variables has been implemented. Its grammar now is :

$$\text{VAR} := \langle \text{LETTER} \rangle / \langle \text{LETTER} \rangle \langle \text{LETTER} \rangle / \langle \text{LETTER} \rangle \langle \text{DIGIT} \rangle$$

Altogether, there are 962 ($= 26 + (26 \times 36)$) allowable variables.

To determine the value of a variable there would be certain memory organization to establish the storage and the accessing sequence. The following approaches will discuss each usage and its efficiency of doing so :

1. Index approach : This method is defined as an index which denotes its position in the table. The conditions existing to apply this method are :

- a. The number of variables is not too large.
- b. The index is easy to compute (example, if the next input symbol is an optional digit d , then the number $26*(d+1)$ is added to the index).
- c. The size of the set is fixed at design time.

This is the method which the Tiny Basic Compiler uses to locate the address for the variable. For the implementation of two characters for the variable's name, this method is not the best one since it requires a fixed number of memory location for storage. Since the less memory used for variable's storage, the more memory could be used for the program, the next method is better in this environment .

2. Linear list approach : The compiler searches through

the program line by line. Each entry will be added to the list if no match to any previously listed name is found. The variable table is organized as a stack of consecutive variable names and storage for each value. The size of this stack is varied depending on the user program. It also can be organized as a linked list which would be useful if the number of variables gets too large. To save memory space and since the number of variable in the I/O basic program is restricted, the stack method is used.

4.1.2.2 CODE GENERATION

The code generation is accomplished via a translator and a coder. The Basic interpreter of the VIC-20 provided an excellent translator since it tokenized all of its identifiers. The control language is implemented with the same process but without using the token for its command.

The coder is a sequence of calling routines to the VIC-20 kernel written in machine code or to the I/O routines which resided in the expansion memory also written in machine code.

The translator maintains the contextual information

that can be derived from the sequence of input tokens. It passes a sequence of commands and its operands to the coder. Each command is interpreted by the coder which also generates appropriate code and then updates the local variable table to reflect the effect of that code.

The coder maintains the run-time contents of the local variables and produces sequences of code to provide the required entries to the VIC-20 primitives. In order to interpret the primitives of the I/O Basic language in terms of the machine code, the coder must maintain description of the values being manipulated (value image) and of the VIC-20 machine environment (machine image). The purpose of the value image is to specify the current representation of each value. Similarly the variable table of the VIC-20 may contain many different values during the course of execution of each variable vector. The relationships between values and the contents of variable vectors are expressed by cross-linkages between the two images (link table).

The following Figure 13 is a flowchart to show the basic sequence algorithm for recognition of an expression of the control language.

4.1.2.3 DETAILED DESCRIPTION OF THE CONTROL LANGUAGE

The translator and the coder communicate by machine language.

1. The translator sets a carry bit or clears a carry bit to indicate direction of the stepper motor. When the coder is activated, it checks this carry bit for the direction function.

2. The translator loads and stores the bit number in the accumulator. The coder will use the contents of the accumulator for relative function.

3. The translator loads the value of a variable and stores its lo, hi address in location 251, 252 (which serves as the link between the translator and the coder) to pass the variable to the coder.

The functions of the control language are described as follows :

Fig 1. INBIT# BIT NUMBER, VARIABLE

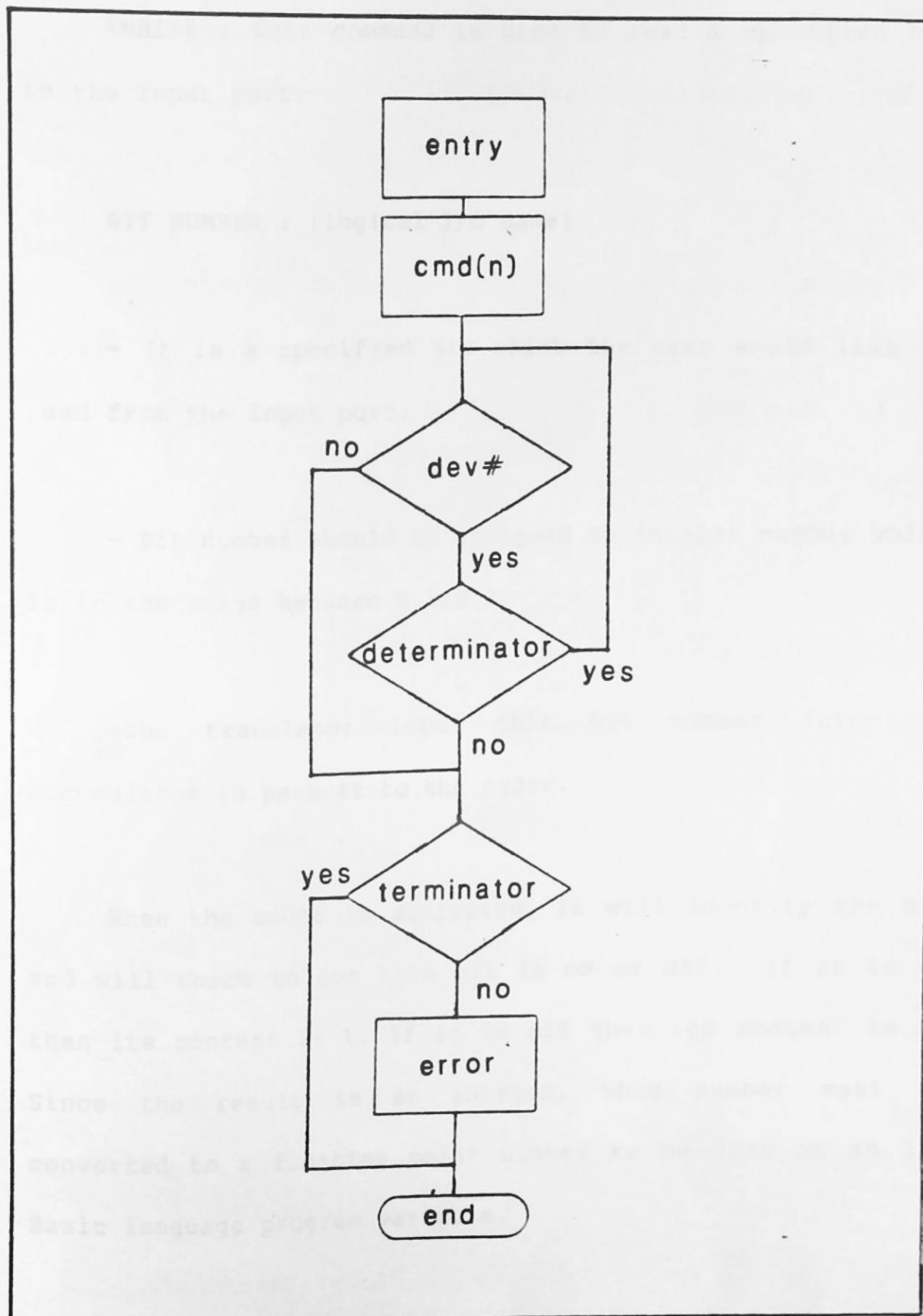


Figure 13:: SEQUENCING FOR RECOGNITION OF AN EXPRESSION

INBIT# : This command is used to read a specified bit in the input port.

BIT NUMBER : (logical I/O name)

- It is a specified bit which the user would like to read from the input port.

- Bit number should be assigned an integer number which is in the range between 0 and 7.

-The translator loads this bit number into the accumulator to pass it to the coder.

When the coder is activated, it will identify the bit and will check to see this bit is on or off. If it is on then its content is 1. If it is off then its content is 0. Since the result is an integer, this number must be converted to a floating point number to be used as an I/O Basic language program variable.

VARIABLE :

The translator passes the variable's Lo, Hi address to the coder by storing the Lo, Hi address of the variable in the locations 251, 252 (link vector).

After built in ROM subroutine IBIT was executed, the result of a specified bit on the input port is stored as a variable in floating point format then passed back to the coder

2.OUTBIT# BIT-NUMBER,VARIABLE

OUTBIT# : This command is used to output the value which is contained in the variable to a specified bit on the output port.

BIT-NUMBER :

- It is a specified bit which the user would like to manipulate on the output port.

- Bit number should be assigned as an integer number which is in the range between 0 and 7.

- The translator loads this bit number into the accumulator to pass it to the coder.

When the coder is activated , it will check the variable for its value ,convert it into integer then apply it to toggle the selected bit on the output port on/off accordingly.

VARIABLE :

- It is used to assign the value which is either 0 (on) or 1 (off) to a specified output bit.

The translator passes the variable's Lo, Hi address to the coder by storing them in the location 251, 252 of page zero of the VIC-20 memory.

When the coder is activated, it will call upon the built in ROM subroutine OBIT to convert the value of the variable from floating point to integer by calling subroutine \$DBA2 and \$D1AA which are built in ROM subroutines of the VIC-20.

3. INITIAL# STEPPER-MOTOR-NUMBER, DIRECTION

INITIAL# : This command initializes a specified stepper motor to its theta zero state with a specified direction. This direction is either counterclockwise (1) or clockwise (0).

STEPPER-MOTOR-NUMBER:

- This number is used to select which stepper motor to be activated. In this system, there is a maximum number of eight stepper motors or four sets with two in each set. The number will range from 1 to 8

- Its value is an integer.

- The translator loads and stores the value of the specified stepper motor into the accumulator to pass it to the coder.

When the coder is activated, the device number will identify the bit to select direction and to read the index .

If the motor is already at theta zero state, the subroutine exits. Otherwise, the stepper motor is stepped until the signal from INDEX is low.

DIRECTION :

- It is used to specify the direction for the stepper motor.

- It should be used as a variable. The value which is assigned to this variable should be an integer, either 1 (counterclockwise) or 0 (clockwise).

- If the direction is counterclockwise then the translator sets the carry bit. Otherwise, the translator clears it. After setting up the carry bit, the translator passes it to the coder for code generation.

4.INITALL#

It is used to initialize all stepper motors to theta zero.

5.MOVE# STEPPER-MOTOR-NUMBER, STEP, DIRECTION

MOVE# : This command is used to move a specified stepper motor a specified number of steps in a specified direction.

STEPPER-MOTOR-NUMBER : see command number 3.

STEP :

- It used to select the number of steps for the stepper motor.

- It should be used as a variable. The value which is assigned to this variable should be an integer, it varies from 0 to 200.

DIRECTION :

- It used to specify the direction for the stepper motor.

- It should be an integer. The value which is assigned to this integer is either 1 (counterclockwise) or 0 (clockwise).

- If the direction is counterclockwise then the translator sets the carry bit. Otherwise, the translator clears the bit. After setting up the carry bit, the translator passes it to the coder for execution.

6.MOVESET# SET-NUMBER, STEP, DIRECTION

The SET-NUMBER, STEP and DIRECTION are used and defined as in the previous command, MOVE#. It is used to move a set of stepper motors with the same specified steps in the same specified direction. One set of stepper motors is defined as two motors.

7.MOVEALL# STEP, DIRECTION

MOVEALL# : This command is used to move all sets of the stepper motors in the system with the same number of steps in the same direction.

STEP, DIRECTION are used and defined as previous command, MOVESET#.

8.START# DC-MOTOR-NUMBER, SPEED

START# : This command is used to start a specified DC motor at specified speed in SPEED variable.

DC-MOTOR-NUMBER :

- This number is used to select which DC motor to be activated. In this system, there are maximum of eight DC motors. Thus DC-MOTOR_NUMBER should be either 1, 2,... ,7 ,8 .

- Its value is an integer.

- The translator loads and stores the value of the specified DC motor into the accumulator and passes it to the coder.

SPEED :

- It is used to define the speed for the DC motor.

- It should be used as a variable. The value which is assigned to it should be an integer where the range is between 0 to 127. If the assigned speed to the DC- MOTOR is greater than 127 then this speed will automatically be adjusted to 127.

- Since this compiler only gives a relative speed output the desired speed to the DC motor could be selected from the driver .

9.STARTALL# DC-MOTOR-NUMBER,SPEED

STARTALL# : This command is used to move all sets of the DC-motors in the system with the same specified speed. For expansion, this system could be set up as four different sets of the DC-motors without adding more VIA. Thus when this command is activated, all eight DC-motors will be moved.

DC-MOTOR-NUMBER : see DC-MOTOR-NUMBER in START# command for detail

SPEED : see SPEED in START# command above for detail

10.STQP# DC_MOTOR_NUMBER

STQP# : This command is used to halt the DC-motor with its speed information remaining the same.

DC-MOTOR-NUMBER : see DC-MOTOR-NUMBER in START# command for detail

11.STQPALL#

STQPALL# : This command is used to halt all sets of DC motors with their speed information remaining the same.

See STARTALL# command for the detail information of the meaning of "ALL" and how it is manipulated.

12.STTIMER# VARIABLE

STTIMER# : This command is issued to initiate the timer to run with TIMOUT recorded with the value in this variable.

Once the TIME reaches TIMOUT, TIME will be set to zero and the timer will be restarted again at zero.

Input : VARIABLE

Output : The real time clock will be initialized at zero then continue to run until TIMOUT is reached. The value is stored in a reserved location called TIMOUT.

13.RTIMER# VARIABLE

RTIMER# : This command is used to read the current time since the last time when the timer was started except when TIMOUT is reached. The reading will be passed by the coder to store into a specified variable table.

Input : VARIABLE

Output : value is stored in the variable. (up to 9999 clocks ; each clock = 1/60 seconds)

14.SAMPLE# VARIABLE

SAMPLE# : This command is used to pass the time interval between successive reading of the analog channels into the reserved location name INTERVAL.

Input : VARIABLE

Output : The value of time interval are stored in INTERVAL

var : number of clocks ; clock = 1/20 seconds

15.SPEED# D.C. MOTOR-NUMBER,VARIABLE

SPEED# : This command allows the user to read the speed of each individual D.C motor speed via the frequency to voltage converter and the analog to digital converter. The interfaces are typical as specified in the hardware interface requirement. This command only gives a relative speed input in order to implement closed loop control. The calibration of the input speed and output speed will be at the user's convenience. The relative speed readings can be manipulated by scaling and or adjusting for any difficulty in tuning the driver circuits.

D.C. MOTOR-NUMBER : as previously defined

VARIABLE : This is the address where the reading will be passed back to the user program routine. The value will be in floating point format to be used by I/O basic.

16. CHANNEL# CHANNEL-NUMBER,VARIABLE

CHANNEL# : This is the command to read the analog inputs from eight (8) analog inputs of the analog to digital conversion.

CHANNEL-NUMBER : This is an integer from 1 to 8 of 8 channels. These logical I/O names do not reflect directly the physical channels of the A/D circuit since the first eight channels are already assigned to eight D.C motor speeds .

VARIABLE : The reading will be stored in the specified variable location in floating point format. Scaling should be considered here to provide adjustability to certain application.

4.2 CAPACITY

The VIC-20 memory expansion allows the user to bring the user memory up to forty-eight Kbytes. Since the I/O Basic Compiler uses the last eight Kbytes to implement its I/O routine, the available memory is reduced to thirty two Kbytes .

4.2.1 MAX-MIN STORAGE REQUIREMENT

The I/O Basic Compiler requires a minimum of 8 Kbytes to start with plus the I/O modules which resides in the highest expansion slot for another 8 Kbytes. The user program could occupy a maximum of 16 Kbytes. Therefore the minimum memory requirement of I/O basic system will require at least 16 Kbytes in expansion (8 Kbytes of RAM and 8 Kbytes of I/O EPROM).

As far as disk or tape storage are concerned, the machine code program and the loader are two additional requirements for a I/O basic to run successfully the full expansion on the VIC-20 is essential.

4.2.2 I/O CAPACITY

The I/O expansion allows two eight Kbytes of memory spaces. This will be plenty to implement a large number of I/O devices. The I/O compiler does not take full advantage of this expansion since the compiler will provide a reasonable amount of I/O devices in its own capacity. The I/O in this system can be activated as follows:

- up to eight D.C motors
- up to eight steppers
- eight switches
- eight relays
- eight analog channels for sensing devices.

These devices, each assigned a logical I/O name to ease the programming effort, has a physical address as specified. The I/O logical name can be expanded with some modification from the I/O module and the change in hardware to reflect the physical address.

The compiler program will be loaded prior to the editing of the user program thus making it very difficult to enter a large program written in I/O basic. The size of the user program will be limited by the maximum of 1000 lines, 962 variables. Each line will be a maximum of 44 characters. The above restriction poses a definite number of storage requirements for tape or disk space since each user program will include the compiler itself. The machine code also will be written to disk or tape and the loader program also takes up storage on these devices.

4.3 LOADER'S SPECIFICATION

The loader program provides user interaction and loads the program to be executed into the specified memory location. It is very short and does not need to be stored in the same disk or tape with the compiler. This gives some advantage for the user to store one loader per machine code program disk or tape to gain access to different programs without swapping disk or tape.

4.3.1 MEMORY REQUIREMENT

The running of machine code routine required a loader and the machine code program by itself. The machine code is a sequential file which varies depending on the complexity of the source routine which may be viewed as a short hand notation of the coded program. The running of a machine code does not require any more than the loader which is so much smaller than the compiler and thus makes a large memory requirement for the loader unnecessary.

4.3.2 I/O PRIMITIVES --- RESERVED MEMORY LOCATIONS

The I/O primitives are those created for the convenience of saving memory during the compiling process. The implementation of these I/O routines in machine language would also reduce the execution time as well as saving a large amount of compiling time due to the use of creating complex calling sequences instead of coding the whole I/O routine one by one over and over for each function.

In addition, the I/O primitives support the multitasking to provide the real time control capability.

4.3.3 I/O PRIMITIVE DESCRIPTION

The following table simplifies the description of I/O primitives:

IBIT	: input bit status	: \$ A025
OBIT	: output bit	: \$ A028
INIT	: init device <A>	: \$ A00A
IALL	: init all	: \$ A00D
MOVE	: move device <A>	: \$ A010
MSET	: move set <A>	: \$ A013
MALL	: move all	: \$ A016
STRT	: start device <A>	: \$ A019
STALL	: start all	: \$ A01C
STOP	: stop device <A>	: \$ A01F
SPALL	: stop all	: \$ A022
STTIM	: set TIMOUT and run at zero	: \$ A02B
RTIME	: read time from last run	: \$ A02E
SAMPL	: set INTERVAL to sample	: \$ A031
SPEED	: read motor speed from device var	: \$ A034
CHAN	: read analog signal from channel var	: \$ A037

4.4 LOADING AND EXECUTION

The loader must be loaded into the user memory via VIC-20 basic loading process then run to create an interactive screen which allows the user to select the particular program to be loaded. This also allows the user to specify a memory location in which the program would be started from.

As the loading process is done the program will run according to the I/O basic specification. The program will exit when the END instruction is encountered. Thus the system will cease all I/O basic execution at this point and return to the VIC-20 basic entry point.

5.0 APPLICATION

This section will be devoted to make some programming practices to help clarify a few differences of the I/O basic from VIC-20 basic:

- The application of I/O control language
- The application of closed loop control
- The application of multitasking program

5.1 OPEN LOOP REAL TIME CONTROL

The example in the following program will apply the I/O basic language to control a D.C motor :

5.1.1 SAMPLE RUN

```

1 REM THIS IS THE PROGRAM IN I/O CONTROL LANGUAGE
2 REM TO PERFORM I/O OPERATION
3 REM AND TO DEMONSTRATE THE OPERATOR INTERFACE FACILITY
4 REM WITH THE KEYBOARD INPUT TO CONTROL THE MOTOR SPEED
5 REM BINARY INPUT CAN BE USED TO SIMULATE LIMIT SWITCH
6 INPUT"DO YOU WANT TO RUN,YES=1,NO=0"/M
7 IFMGOTO9
8 END
9 INPUT"VELOCITY = "/VI
10 LETVO=1
11 START#1,VO
12 VO=VO+1
13 VA=VI-VO
14 IFVAGOTO11
21 INBIT#1,IB
22 OUTBIT#1,IB
23 IFIBGOTO9
24 MOVE#2,VI,1
25 STOP#1:GOTO6

```

READY.

5.1.2 EFFICIENCY AND EXPANSION

The sample program has the convenience of issuing a single command to control the motor to start and to stop. It also allows the speed to be changed with any function at user advantage. The degree of freedom introduced here can be considered the main advantage over many programmable controllers which have only a few variables and computational power. In spite of the fact of having so much flexibility, the resolution of 128 for the motor speed would not be a user delight.

5.2 CLOSED LOOP REAL TIME CONTROL

The following is a sample program of a closed loop control to demonstrate the flexibility of the I/O basic language.

5.2.1 SAMPLE RUN

```
1 REM THIS IS THE PROGRAM IN I/O CONTROL LANGUAGE
2 REM TO PERFORM CLOSED LOOP CONTROL
3 REM AND TO DEMONSTRATE THE OPERATOR INTERFACE FACILITY
4 REM WITH THE KEYBOARD INPUT TO CONTROL THE MOTOR SPEED
5 REM BINARY INPUT CAN BE USED TO SIMULATE LIMIT SWITCH
6 INPUT "DO YOU WANT TO RUN,YES=1,NO=0"/M
7 IFMGOTO9
8 END
9 INPUT "SAMPLE = "/SA
10 INPUT "SETTIME = "/ST
11 INPUT "SPEED = "/VI
12 SAMPLE#SA
13 STTIMER#ST
14 START#1,VI
15 VN=VI
16 RTIMER#TI
17 IFTI GOTO16
21 INBIT#1,IB
22 OUTBIT#1,IB
23 IFIBGOTO9
24 SPEED#1,VO
25 VA=VI-VO
26 VB=ABS(VA)
27 VC=VA-VB
28 IFVC GOTO40
29 VN=VN+1
30 START#1,VN
31 INBIT#2,IB
32 OUTBIT#2,IB
33 IFIBGOTO16
34 STOP#1:GOTO6
40 VN=VN-1:GOTO30
```

```

1 REM THIS IS THE PROGRAM IN I/O CONTROL LANGUAGE
2 REM TO PERFORM CLOSED LOOP CONTROL
3 REM AND TO DEMONSTRATE THE OPERATOR INTERFACE FACILITY
4 REM WITH THE KEYBOARD INPUT TO CONTROL THE MOTOR SPEED
5 REM BINARY INPUT CAN BE USED TO SIMULATE LIMIT SWITCH
6 INPUT"DO YOU WANT TO RUN,YES=1,NO=0"/M
7 IFMGOTO9
8 END
9 INPUT"VELOCITY = "/VI
10 SPEED#1,VO
11 START#1,VO
12 VO=VO+1
13 VH=VI-VO
14 IFVAGOTO11
15 VH=VI
21 INBIT#1,IB
22 OUTBIT#1,IB
23 IFIBGOTO9
24 SPEED#1,VO
25 VH=VI-VO
26 VE=ABS(VR)
27 VC=VH-VE
28 IFVCGOTO40
29 VN=VN+1
30 START#1,VN
31 INBIT#2,IB
32 OUTBIT#2,IB
33 IFIBGOTO24
34 STOP#1:GOTO6
40 VN=VN-1:GOTO30

```

```

1 REM THIS IS THE PROGRAM IN I/O CONTROL LANGUAGE
2 REM TO PERFORM CLOSED LOOP CONTROL
3 REM AND TO DEMONSTRATE THE OPERATOR INTERFACE FACILITY
4 REM WITH THE KEYBOARD INPUT TO CONTROL THE MOTOR SPEED
5 REM BINARY INPUT CAN BE USED TO SIMULATE LIMIT SWITCH
6 INPUT"DO YOU WANT TO RUN,YES=1,NO=0"JM
7 IFMGOTO9
8 END
9 INPUT"SAMPLE = ";SA
10 INPUT"SETTIME =";ST
11 INPUT"SPEED = ";VI
12 SAMPLE#SA
13 STTIMER#ST
14 START#1,VI
15 VN=VI
16 VJ=127-VI:VM=VJ
17 START#2,VJ
18 RTIMER#TI
19 IFTIGOTO16
20 INBIT#1,IB
21 OUTBIT#1,IB
23 IFIBGOTO9
24 SPEED#1,VO
25 SPEED#2,VP
26 VA=VI-VO
27 VB=VJ-VP
28 VN=VI+VA:REMPROPORTIONAL
29 VM=VI+VB:REMCONTROL
30 VN=ABS(VN):REMIF
31 VM=ABS(VM):REMT00 SMALL INPUTS COULD BE NEGATIVE
32 START#1,VN
33 START#2,VM
34 STOP#1:GOTO6
41 INBIT#2,IB
42 OUTBIT#2,IB
43 IFIBGOTO24
44 STOP#1
45 STOP#2
46 GOTO6

```

5.2.2 EFFICIENCY AND EXPANSION

The closed loop control in this example is a proportional control with sampling time equal to 2 seconds and the motor is allowed to run for a period of 30 minutes. This control policy can be implemented as integral, PI or PID at the user strategies. Taking advantage of :

STIMER# VAR

RTIMER# VAR and

SAMPLE# VAR

The real time control can be realized in a simple manner and the sampling time to be implemented as a variable could be used in variable sampling time application. The same advantage is also applied to the time to run of the motor. These advantages give a tremendous amount of freedom to the control policy once implemented by the user.

5.3 MULTITASKING

The multitasking facility in the compiler utilizes 1/60 second time-slices to perform the background task such as :

- display refreshing
- keyboard scanning
- time updating
- motor stepping
- channel data reading

Thus, commands such as MOVE#, SPEED# and CHANNEL# are no more than writing or reading data to or perform a memory location, it only takes less than 0.1 msec compared with the "real world" delay of each step of 10 msec or more and the analog to digital conversion delay of each reading of 1 msec.

The following program is an example of multitasking via the flexibility of the I/O control language:

```

50 INPUT STEP #VST
51 MOVE#2,ST,1
52 ST=200-ST
53 MOVE#1,ST,1
54 CHANNEL#1,CH
55 PRINT "VALUE = ",CH
56 GOTO 50

```

5.3.1 SAMPLE RUN

```

1 REM THIS IS THE PROGRAM IN I/O CONTROL LANGUAGE
2 REM TO PERFORM MULTITASKING IN CONTROL
3 REM AND TO DEMONSTRATE THE OPERATOR INTERFACE FACILITY
4 REM WITH THE KEYBOARD INPUT TO CONTROL THE ENTIRE PROCESS
5 REM BINARY INPUT CAN BE USED TO SIMULATE LIMIT SWITCH
6 INPUT"DO YOU WANT TO RUN,YES=1,NO=0";M
7 IFMGOTO9
8 END
9 INPUT"SAMPLE = ";SA
10 INPUT"SETTIME = ";ST
11 INPUT"SPEED = ";VI
12 SAMPLE#SA
13 STTIMER#ST
14 START#1,VI
15 VN=VI
16 VJ=127-VI:VM=VJ
17 START#2,VJ
19 RTIMER#TI
20 IFTIGOTO16
21 INBIT#1,IB
22 OUTBIT#1,IB
23 IFIBGOTO50
24 SPEED#1,VO
25 SPEED#2,VP
26 VA=VI-VO
27 VB=VJ-VP
28 VN=VI+VA:REMPROPORTIONAL
29 VM=VI+VB:REMCONTROL
30 VN=ABS(VN):REMIF
31 VM=ABS(VM):REMT00 SMALL INPUTS COULD BE NEGATIVE
32 START#1,VN
33 START#2,VM
34 STQP#1:GOTO6
41 INBIT#2,IB
42 OUTBIT#2,IB
43 IFIBGOTO19
44 STQP#1
45 STQP#2
46 GOTO6
50 INPUT"STEP = ";ST
51 MOVE#2,ST,1
52 ST=200-ST
53 MOVE#1,ST,1
54 CHANNEL#1,CH
55 PRINT"VALUE = ";CH
56 GOTO9

```

5.3.2 EFFICIENCY AND EXPANSION

The above program demonstrates the capability of the I/O basic to implement a multitasking procedure which controls two motors at the same time in a closed loop configuration. The complexity of this control must be realized from the user standpoint. Of course this grammar does seem to have some unusual applications and difficult to be realized particularly because the limitations in term of mathematical functions. The conditional statement is also not very convenient although it may have had some programming power to it.

6.0 CONCLUSIONS

The I/O Basic Compiler in this thesis is an experiment of putting a low cost device to work at a minicomputer level. The real issue of where can we draw the line between a simple control system and a more complex one and how can we find realistically a limitation of a machine with its own original architecture and some additional peripherals is not particularly clear.

First , let us take a look at the unconventional editing, loading and execution of the I/O basic compiler.

The editing of a user program turns out to be fairly restricted, and the program length is also limited because of the compiler program which occupies almost the entire memory space for the VIC-20 user portion.

There could have been an easy way to solve this problem by allowing the user to edit his/her program in the entire space of the allowable VIC-20 user memory then store it in

disk or tape. The compiler then can be activated to load a segment of the user routine in and to compile it, one segment at a time. Each segment of machine code will be written onto disk space or tape one segment at a time. Thus the entire user program can be edited with very small memory usage.

The loading of the machine code is provided by the loader which resides in a different program and thus makes the sequence of operating more difficult to maintain since the memory allocation for the machine code must be taken into account. If the user is not aware of the existing difficulty of his/her task when assigning a particular memory location for his/her program devastating results could be imminent.

The editing process also creates an irritating problem because the compiler, since it occupies user memory, could be destroyed by any editing error.

Second, the I/O Basic routine always uses an integer number for its execution while the user program variables

are always stored in the floating point format. This incompatibility requires a large number of base-conversion routines thus jeopardizing the execution speed. In spite of the fact that such an implementation would give a tremendous flexibility to the user program the variable type can be used to advantage by a specifically designated integer variable table. The only requirement for this is the extra expressions for integer variables and it could be converted to floating point at user convenience.

Third , the multitasking feature of this machine has a unique background task that allows a VIC-20 to be able to perform multi-tasking efficiently. The architecture could be enhanced to provide some pre-emptive actions for priority handling of critical task interruptions. The discussion of the multitasking of such a degree are beyond the scope of this thesis .

Although some disadvantages and difficulties of the I/O basic Compiler exist, the I/O Basic Compiler has achieved a very interesting application of the VIC-20.

First, considering the ease of writing an I/O routine with simple single I/O instructions as a programmable controller but also having the ease of sequencing these routines with every available variable, a feature which no programmable controller ever offered. For example , the programmable controller can be :

```
START #1 , SPEED = 20 RPM .....
```

```
IF (condition) CHANGE SPEED
```

The condition is fixed and difficult to establish since no computation can be performed.

But the I/O Basic Compiler would have the following :

```
LET A = 20
```

```
nn START #1 , SPEED = A (to initialize at this speed)
```

```
A = (any desired function to dynamically adjust the  
speed)
```

```
IF (condition) GOTO nn
```

Thus the closed loop control can be implemented on the proposed system with greater freedom.

Second , the I/O basic compiler has a significant advantage for the user to use the I/O routine throughout the entire program with no restriction in terms of programming language . This feature makes the VIC-20 basic interpreter obsolete because POKE, PEEK, USR(n) and SYS(n) can not be so useful and meaningful since it requires the user to be completely responsible for every detail of control sequence in his/her routine and still can not achieve the flexibility of all variables and the speed of execution .

Third , the I/O basic compiler also offers an extension toward a more complex and powerful application if it is utilized to handle a part of a larger system . This will require additional routines to handle the communication which can be drawn from the existing VIC-20 kernel with additional communication hardware to transfer data from one to another. With dedicated I/O and dedicated buffers the closely-coupled net via its IEEE-488 serial bus can be as powerful as any minicomputer in distributed control systems.

APPENDIX A

A.1 I/O BUS SPECIFICATIONS

APPENDIX

SOFTWARE FEATURES

- External power supply connection
- Directly connected to the Data bus
- Addressable I/O block
- Up to 8 I/O interface modules to be supported

INPUT SPECIFICATIONS

- Fully buffered data bus, TTL compatible
- External power supply 5 VDC $\pm 5\%$ to $\pm 10\%$
- Interrupt, reset input signal available

OUTPUT SPECIFICATIONS

- 8 selectable I/O range via address

APPENDIX A

A.1 I/O BUS SPECIFICATIONS

HARWARE FREATURE

- External power supply connection
- Directly connected to the data bus
- Addressable I/O block
- Up to 8 I/O interface modules to be connected

INPUT SPECIFICATIONS

- Fully buffered data bus. TTL compatible
- External power supply 5 VDC -5% to +5%
- Interrupt, reset input signal available

OUTPUT SPECIFICATIONS

- 8 selectable I/O range via minijumper

- TTL compatible
- 02, R/W signals available

A.2 UNIVERSAL INTERFACE MODULE

HARDWARE FEATURES

- Eight opto-isolated digital monitoring channels
- Eight forms A relay outputs
- On-board transient protection circuitry

INPUT SPECIFICATIONS

- Eight opto-isolated digital input channels
- Minimum input isolation : 300 volts
- Maximum continuous input voltage : 24 volts
- Two opto-isolated handshake input signals
- Minimum input isolation : 300 volts
- Maximum input voltage : 5 volts

OUTPUT SPECIFICATIONS

- Maximum switching : 30 volts DC or 120 volts AC
0.5 amperes

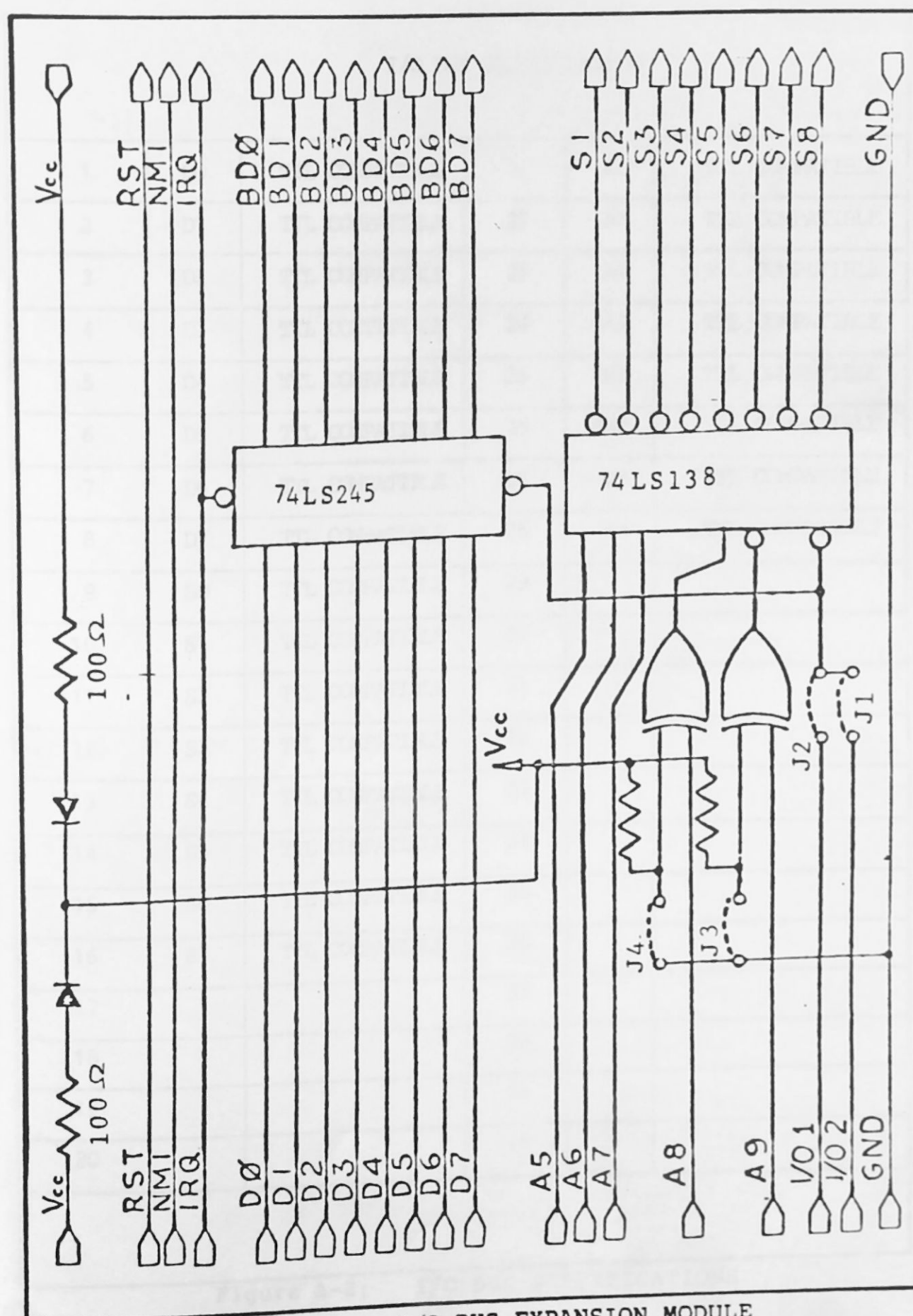


Figure A-1: I/O BUS EXPANSION MODULE

I/O BUS SPECIFICATION					
1	DO	TTL COMPATIBLE	21	AO	TTL COMPATIBLE
2	D1	TTL COMPATIBLE	22	A1	TTL COMPATIBLE
3	D2	TTL COMPATIBLE	23	A2	TTL COMPATIBLE
4	D3	TTL COMPATIBLE	24	A3	TTL COMPATIBLE
5	D4	TTL COMPATIBLE	25	INT	TTL COMPATIBLE
6	D5	TTL COMPATIBLE	26	RST	TTL COMPATIBLE
7	D6	TTL COMPATIBLE	27	O2	TTL COMPATIBLE
8	D7	TTL COMPATIBLE	28	RW	TTL COMPATIBLE
9	SO	TTL COMPATIBLE	29		
10	S1	TTL COMPATIBLE	30		
11	S2	TTL COMPATIBLE	31		
12	S3	TTL COMPATIBLE	32		
13	S4	TTL COMPATIBLE	33		
14	S5	TTL COMPATIBLE	34		
15	S6	TTL COMPATIBLE	35		
16	S7	TTL COMPATIBLE	36		
17			37		
18			38		
19			39		
20			40		

Figure A-2: I/O BUS SPECIFICATIONS

10 VA resistive

- Minimum isolation : 300 volts

transient protectors (Tranzorbs) are
provided for reactive loads

- Two opto-isolated handshake output signals
- Minimum isolation : 300 volts
- Maximum switching voltage : 30 volts DC (100 mA)

A.3 STEPPER MOTOR INTERFACE (INPUTS)

HARDWARE FEATURES

- 16 opto-isolated digital monitoring channels
- TTL compatible
- On-board transient protection circuitry

INPUT SPECIFICATIONS

- 16 opto-isolated digital input channels
- Minimum input isolation : 300 volts
- Maximum continuous input voltage : 24 volts

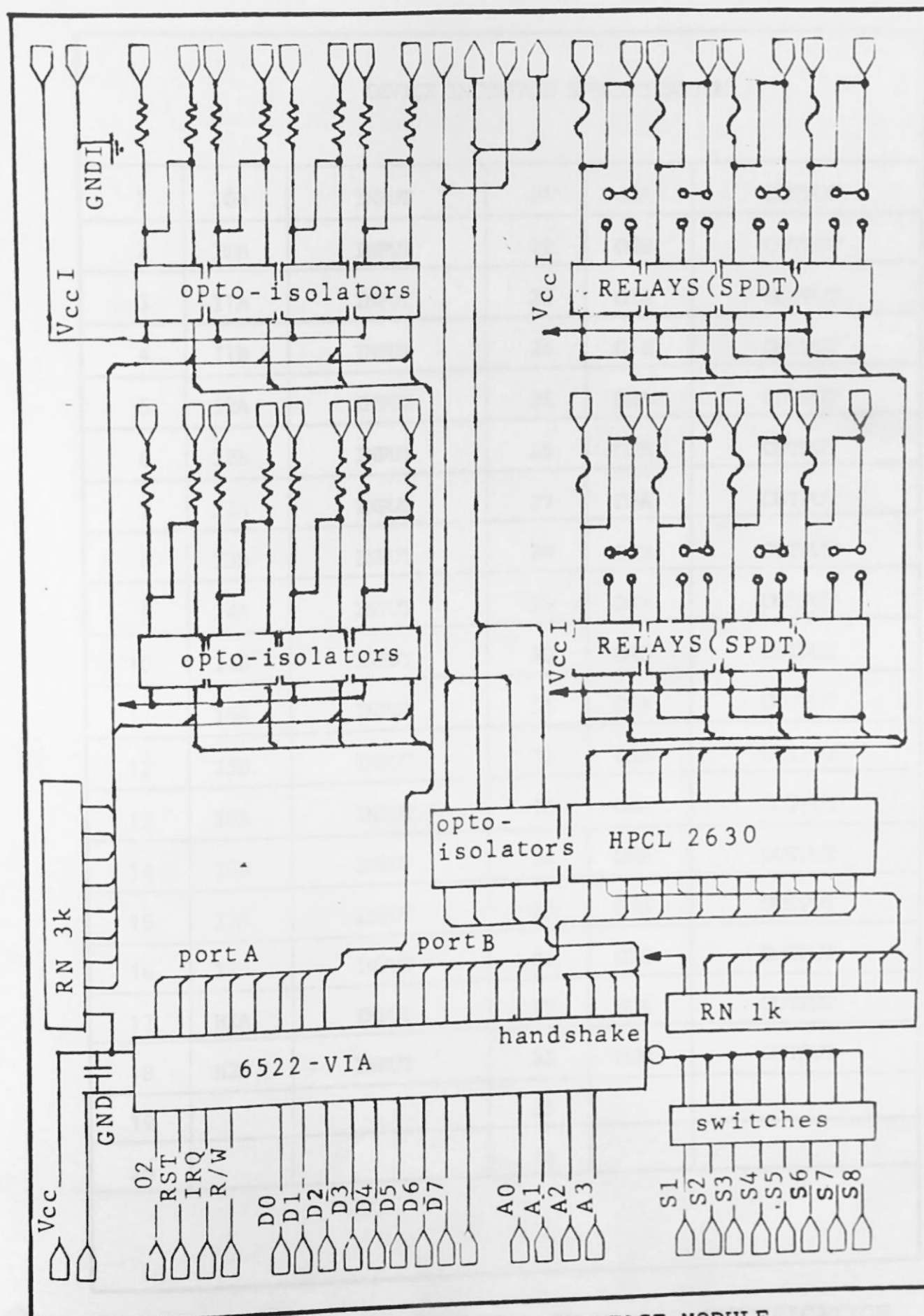


Figure A-3: UNIVERSAL INTERFACE MODULE

DEVICE INTERFACE SPECIFICATION					
1	I0A	INPUT	21	O0A	OUTPUT
2	I0B	INPUT	22	O0B	OUTPUT
3	I1A	INPUT	23	O1A	OUTPUT
4	I1B	INPUT	24	O1B	OUTPUT
5	I2A	INPUT	25	O2A	OUTPUT
6	I2B	INPUT	26	O2B	OUTPUT
7	I3A	INPUT	27	O3A	OUTPUT
8	I3B	INPUT	28	O3B	OUTPUT
9	I4A	INPUT	29	O4A	OUTPUT
10	I4B	INPUT	30	O4B	OUTPUT
11	I5A	INPUT	31	O5A	OUTPUT
12	I5B	INPUT	32	O5B	OUTPUT
13	I6A	INPUT	33	O6A	OUTPUT
14	I6B	INPUT	34	O6B	OUTPUT
15	I7A	INPUT	35	O7A	OUTPUT
16	I7B	INPUT	36	O7B	OUTPUT
17	H1A	INPUT	37	HOA	OUTPUT
18	H2B	INPUT	38	HOB	OUTPUT
19			39		
20			40		

Figure A-4: UNIVERSAL INTERFACE SPECIFICATION

- Minimum input voltage : 0 volts

A.4 STEPPER MOTOR INTERFACE (OUTPUTS)

HARDWARE FEATURES

- 16 Binary output channels
- TTL compatible

OUTPUT SPECIFICATIONS

- Maximum switching : 30 volts DC
100 mA
- Minimum isolation : 300 volts

A.5 DC MOTOR INTERFACE MODULE

HARDWARE FEATURES

- Eight analog output channels
- Seven bits resolution
- Industry standard voltage or current input
- On-board transient protection circuitry

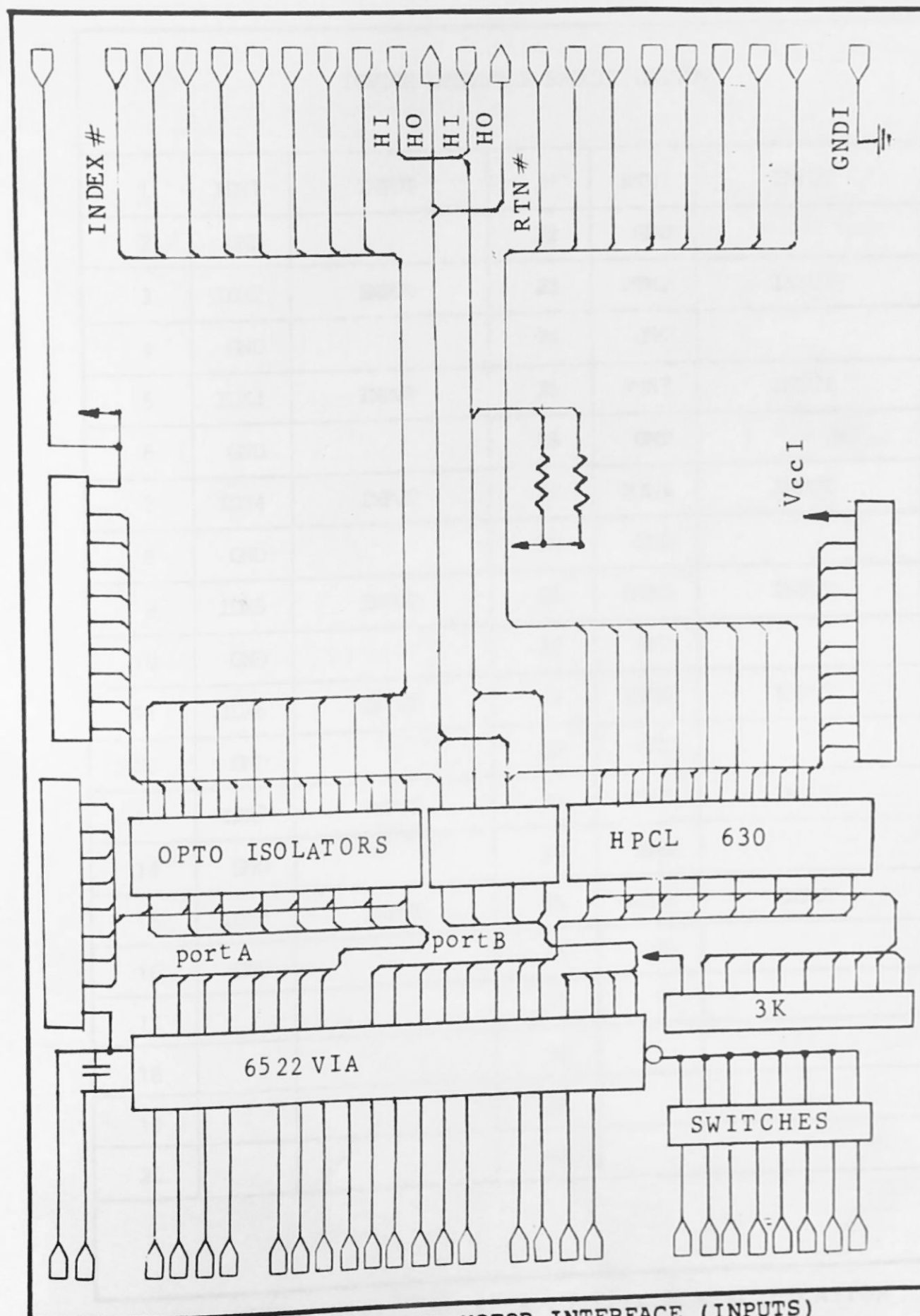


Figure A-5: STEPPER MOTOR INTERFACE (INPUTS)

DEVICE INTERFACE SPECIFICATION					
1	IDX1	INPUT	21	RIN1	INPUT
2	GND		22	GND	
3	IDX2	INPUT	23	RIN2	INPUT
4	GND		24	GND	
5	IDX3	INPUT	25	RIN3	INPUT
6	GND		26	GND	
7	IDX4	INPUT	27	RIN4	INPUT
8	GND		28	GND	
9	IDX5	INPUT	29	RIN5	INPUT
10	GND		30	GND	
11	IDX6	INPUT	31	RIN6	INPUT
12	GND		32	GND	
13	IDX7	INPUT	33	RIN7	INPUT
14	GND		34	GND	
15	IDX8	INPUT	35	RIN8	INPUT
16	GND		36	GND	
17			37		
18			38		
19			39		
20			40		

Figure A-6: STEPPER INTERFACE SPECIFICATION

DEVICE INTERFACE SPECIFICATION					
1	DIR1	OUTPUT	21	STP1	OUTPUT
2	GND	OUTPUT	22	GND	OUTPUT
3	DIR2	OUTPUT	23	STP2	OUTPUT
4	GND	OUTPUT	24	GND	OUTPUT
5	DIR3	OUTPUT	25	STP3	OUTPUT
6	GND	OUTPUT	26	GND	OUTPUT
7	DIR4	OUTPUT	27	STP4	OUTPUT
8	GND	OUTPUT	28	GND	OUTPUT
9	DIR5	OUTPUT	29	STP5	OUTPUT
10	GND	OUTPUT	30	GND	OUTPUT
11	DIR6	OUTPUT	31	STP6	OUTPUT
12	GND	OUTPUT	32	GND	OUTPUT
13	DIR7	OUTPUT	33	STP7	OUTPUT
14	GND	OUTPUT	34	GND	OUTPUT
15	DIR8	OUTPUT	35	STP8	OUTPUT
16	GND	OUTPUT	36	GND	OUTPUT
17			37		
18			38		
19			39		
20			40		

Figure A-8: STEPPER INTERFACE SPECIFICATION

- Optically Isolated Interface

between processor and driver modules

OUTPUT SPECIFICATIONS

- isolated output channels
- Output ranges : 0 to V_{pp} (up to 28 VDC)
- Resolution : 7 -bits binary
- Accuracy : relative $-1/4$ or $+1/4$ LSB
fullscale -1 or $+1$ LSB
Zero error -1 or $+1$ LSB
- Power Supply Range : $V_{cc} = 5$ VDC, $V_{pp} = 5$ to 30 VDC
- Compliance Voltage : (power supply voltage) -10 volts
- Isolation : 300 volts

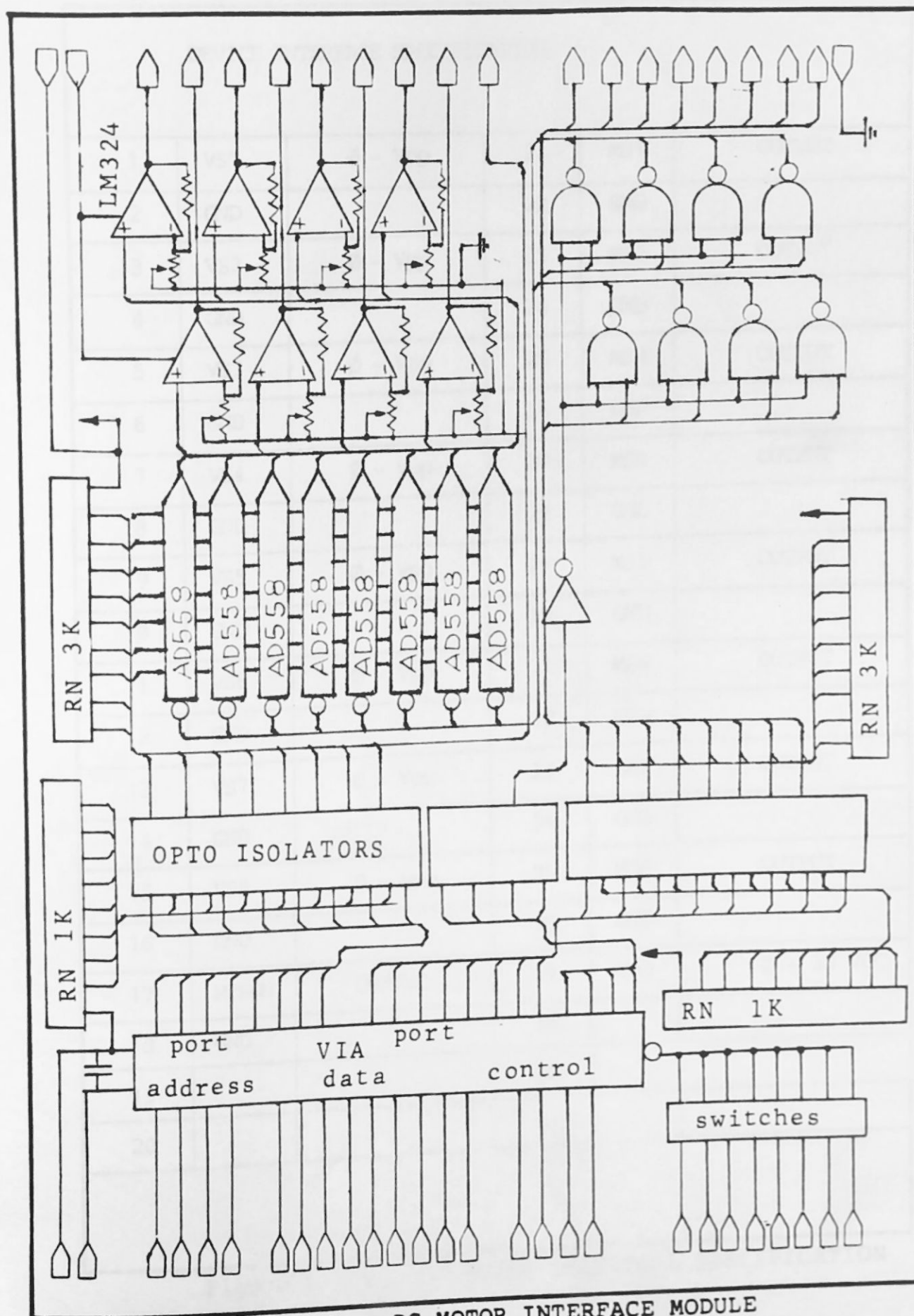


Figure A-9: DC MOTOR INTERFACE MODULE

DEVICE INTERFACE SPECIFICATION					
1	VS1	$\emptyset - V_{pp}$	21	MS1	OUTPUT
2	GND		22	GND	
3	VS2	$\emptyset - V_{pp}$	23	MS2	OUTPUT
4	GND		24	GND	
5	VS3	$\emptyset - V_{pp}$	25	MS3	OUTPUT
6	GND		26	GND	
7	VS4	$\emptyset - V_{pp}$	27	MS4	OUTPUT
8	GND		28	GND	
9	VS5	$\emptyset - V_{pp}$	29	MS5	OUTPUT
10	GND		30	GND	
11	VS6	$\emptyset - V_{pp}$	31	MS6	OUTPUT
12	GND		32	GND	
13	VS7	$\emptyset - V_{pp}$	33	MS7	OUTPUT
14	GND		34	GND	
15	VS8	$\emptyset - V_{pp}$	35	MS8	OUTPUT
16	GND		36	GND	
17	MTRON	OUTPUT	37	V_{pp}	$\emptyset - 30 \text{ VDC}$
18	GND		38		
19			39		
20			40		

Figure A-10: DC MOTOR INTERFACE SPECIFICATION

A.6 ANALOG INPUTS INTERFACE MODULE

HARDWARE FEATURES

- Two groups of 8 single-ended analog monitoring channels
- Seven bit resolution
- Industry standard voltage or current input
- Integrating inputs for optimum power line noise rejection
- Each channel converted every 400 microseconds
- On-board transient protection circuitry
- Optically Isolated Interface

between processor and analog to digital converter DAS-952R

INPUT SPECIFICATIONS

- 16 single-ended input channels
- Input range : 0-5 VDC, 0-20mA
- Resolution : 8-bits binary
- Accuracy : 7-bits binary
- Temperature Coefficient of Accuracy

Voltage Range : -0.003% or +0.003% per Degree C :

Current Range : -0.008% per Degree C

- Common Mode Rejection Ratio : 86 dB
- Voltage Input Configuration
 - maximum common mode input voltage :
 - 5 volts for maximum accuracy
 - 30 volts without damage

A.7 STEPPER MOTOR DRIVER MODULE A/B

HARDWARE FEATURES

- Programmable stepping rate
- Fullwave drive
- Module A for bipolar drive
- Module B for unipolar chopper drive
- Up to 0.5 A per phase
- Up to 30 VDC supply
- On board protection circuitry

INPUT SPECIFICATIONS

- TTL compatible inputs
- External power supply up to 30 VDC

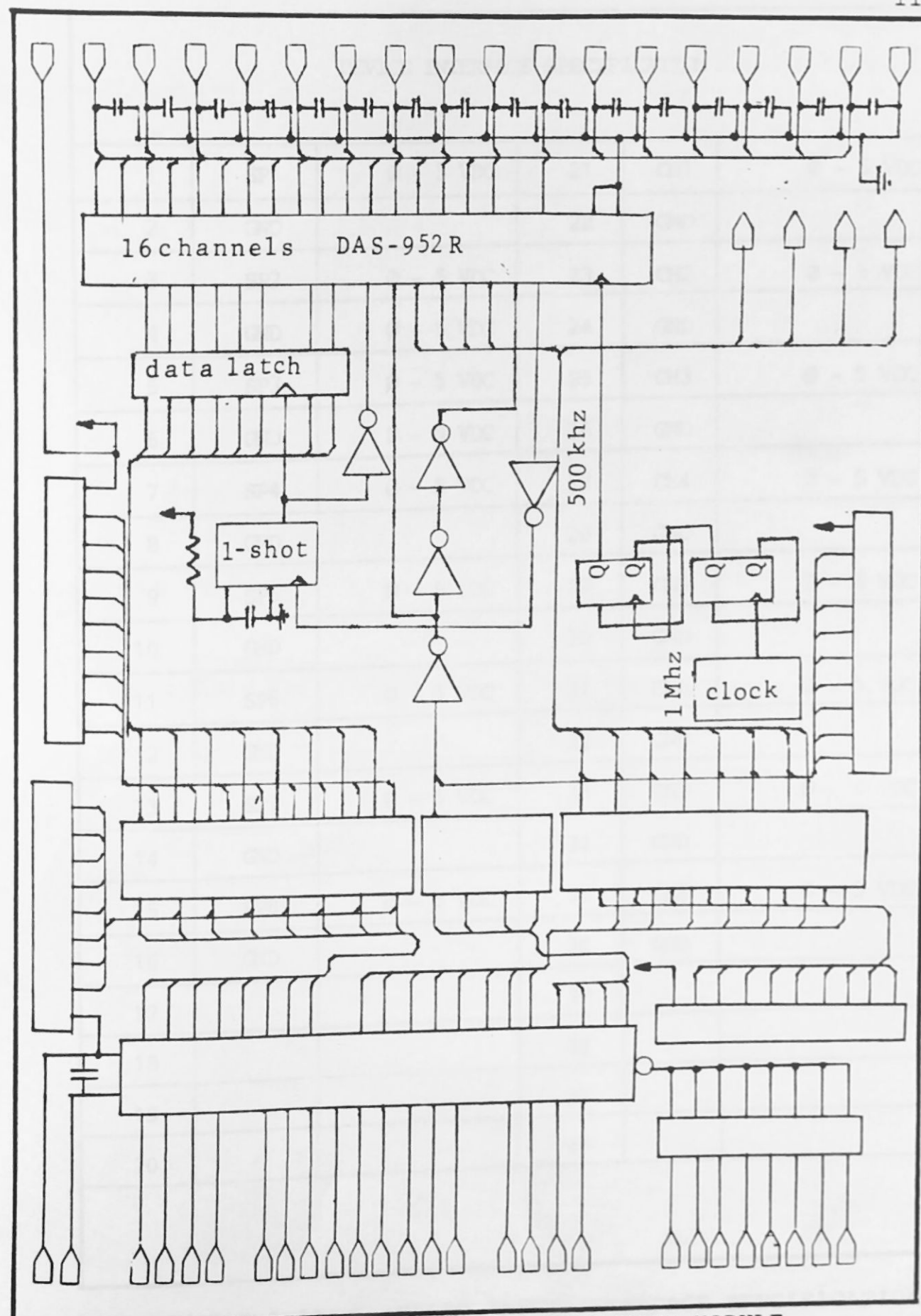


Figure A-11: ANALOG INPUT INTERFACE MODULE

DEVICE INTERFACE SPECIFICATION					
1	SP1	Ø - 5 VDC	21	CH1	Ø - 5 VDC
2	GND		22	GND	
3	SP2	Ø - 5 VDC	23	CH2	Ø - 5 VDC
4	GND	Ø - 5 VDC	24	GND	
5	SP3	Ø - 5 VDC	25	CH3	Ø - 5 VDC
6	GND	Ø - 5 VDC	26	GND	
7	SP4	Ø - 5 VDC	27	CH4	Ø - 5 VDC
8	GND		28	GND	
9	SP5	Ø - 5 VDC	29	CH5	Ø - 5 VDC
10	GND		30	GND	
11	SP6	Ø - 5 VDC	31	CH6	Ø - 5 VDC
12	GND		32	GND	
13	SP7	Ø - 5 VDC	33	CH7	Ø - 5 VDC
14	GND		34	GND	
15	SP8	Ø - 5 VDC	35	CH8	Ø - 5 VDC
16	GND		36	GND	
17			37		
18			38		
19			39		
20			40		

Figure A-12: ANALOG INPUT INTERFACE SPECIFICATION

OUTPUT SPECIFICATIONS

- module A : up to 0.5 A per phase
- module B : up to 0.5 A per phase
- chopper output voltage available

A.8 DC MOTOR DRIVER MODULE

HARDWARE FEATURES

- Programmable speed range
- Current feedback for stability
- Dynamic braking
- Motorjam detection
- On board protection circuitry
- Up to 5 A surge current
- Stop and start operation

INPUT SPECIFICATIONS

- Analog input for speed control (0 to 30 VDC)
- External power supply up to 30 VDC

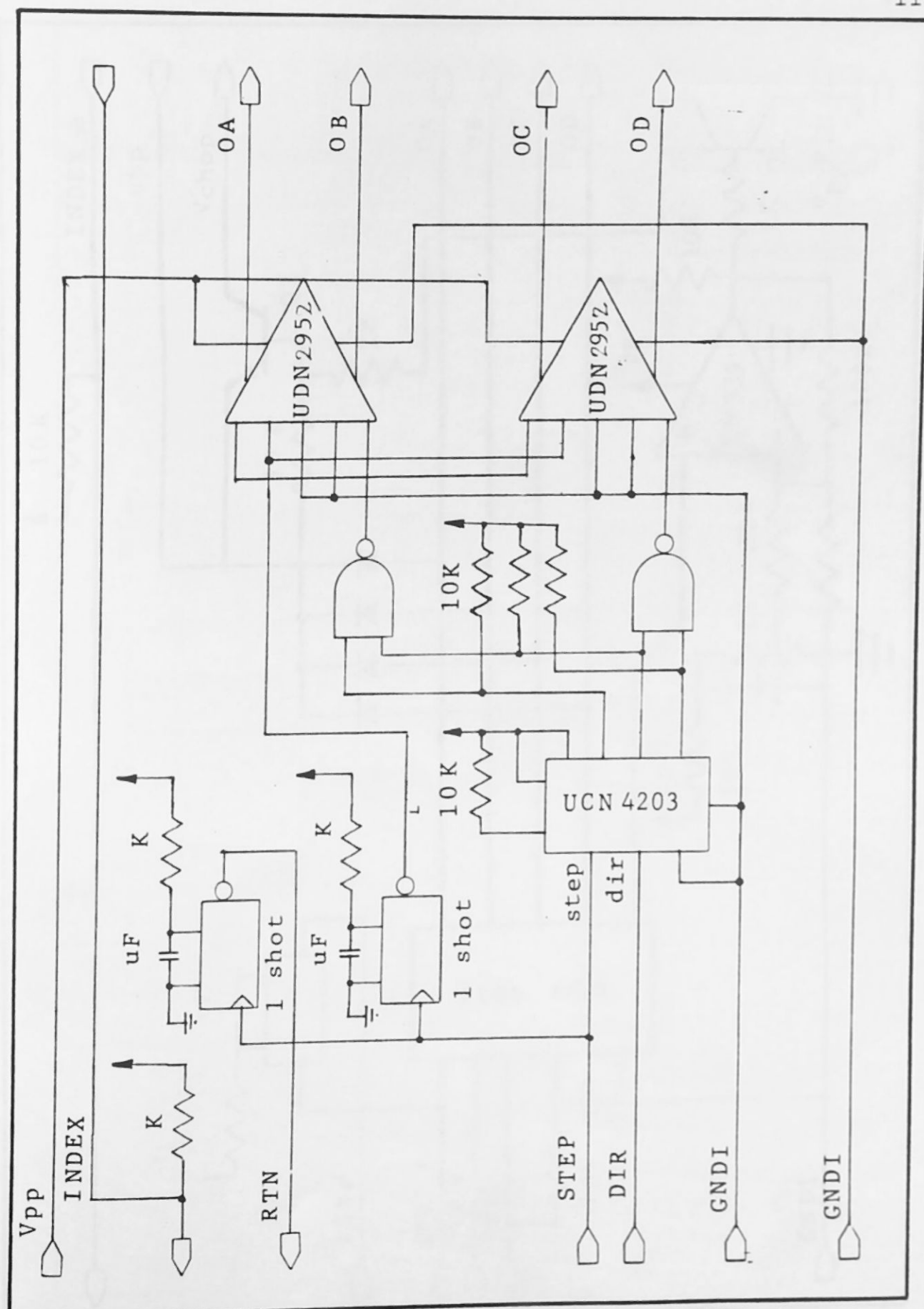


Figure A-13: STEPPER MOTOR DRIVER MODULE A

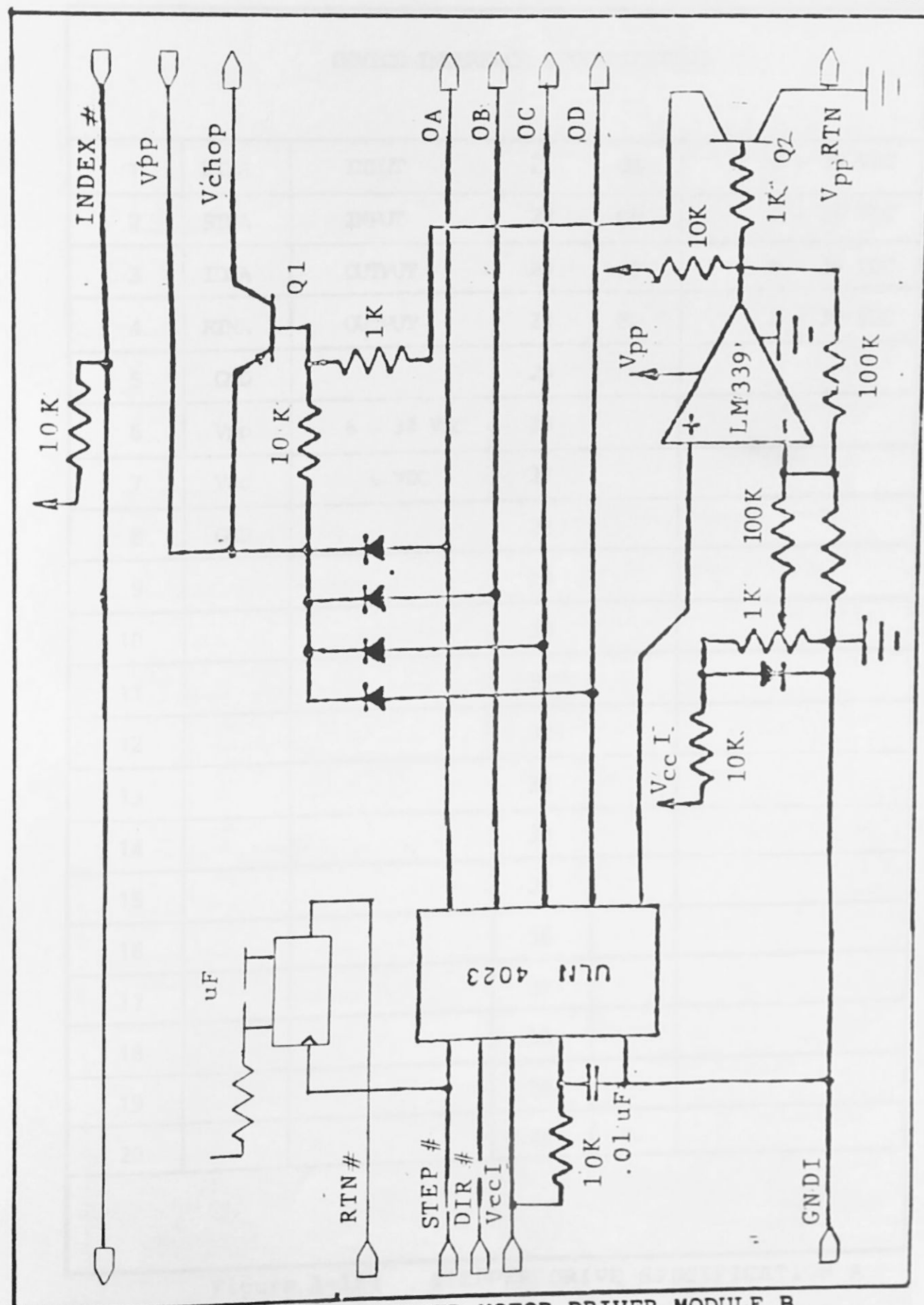


Figure A-14: STEPPER MOTOR DRIVER MODULE B

DEVICE INTERFACE SPECIFICATION					
1	DIRA	INPUT	21	ØA	Ø - 3Ø VDC
2	STPA	INPUT	22	ØB	Ø - 3Ø VDC
3	IDXA	OUTPUT	23	ØC	Ø - 3Ø VDC
4	RTNA	OUTPUT	24	ØD	Ø - 3Ø VDC
5	GND		25		
6	Vpp	6 - 3Ø VDC	26		
7	Vcc	5 VDC	27		
8	GND		28		
9			29		
10			30		
11			31		
12			32		
13			33		
14			34		
15			35		
16			36		
17			37		
18			38		
19			39		
20			40		

Figure A-15: STEPPER DRIVE SPECIFICATION A

DEVICE INTERFACE SPECIFICATION					
1			21		
2			22		
3			23		
4			24		
5			25		
6			26		
7			27		
8			28		
9			29		
10			30		
11	DIRB	INPUT	31	OA	0 - 30 VDC
12	STPB	INPUT	32	OB	0 - 30 VDC
13	IDXB	INPUT	33	OC	0 - 30 VDC
14	RTNB	OUTPUT	34	OD	0 - 30 VDC
15	GND		35	Vchop	0 - 30 VDC
16	Vpp	6 - 30 VDC	36		0 - 30 VDC
17	Vcc	5 VDC	37		
18	GND		38		
19			39		
20			40		

Figure A-16: STEPPER DRIVE SPECIFICATION B

- TTL compatible control input

OUTPUT SPECIFICATIONS

- Up to 5A/30 VDC
- TTL compatible motor jammed output

A.9 SPEED SENSING MODULE

HARDWARE FEATURES

- To be used with photo sensing or hall effect sensing
- Dynamic threshold control
- Adjustable frequency input range
- Adjustable output voltage range
- TTL compatible
- Frequency output

INPUT SPECIFICATIONS

- Analog input : 0-5 VDC
- Common mode rejection ratio : 86 dB

DEVICE INTERFACE SPECIFICATION					
1	MITRON	INPUT	21	DCM+	0 - Vpp
2	GND		22	Vpp	0 - 30 VDC
3	VS	0 - Vpp	23	DMC-	0 - 1 VDC
4	GND		24	GND	
5	MS	INPUT	25		
6	MJAM	OUTPUT	26		
7			27		
8			28		
9			29		
10			30		
11			31		
12			32		
13			33		
14			34		
15			35		
16			36		
17			37		
18			38		
19			39		
20			40		

Figure A-18: DC MOTOR DRIVE SPECIFICATION

- Maximum frequency range : 5 kHz
- Minimum frequency range : 0 Hz

OUTPUT SPECIFICATIONS

- Analog output : 0-5 VDC, 0-20 mA

A.10 INPUT CONDITIONING MODULE

HARDWARE FEATURE

- AC or DC measurements
- Full wave rectified AC inputs
- Scaling for high input voltages
- On board transient protection

INPUT SPECIFICATIONS

- Analog input : 0-30 V (AC or DC)
- Common mode rejection ratio : 86 dB

OUTPUT SPECIFICATIONS

- Analog output : 0-5 VDC, 0-20 mA

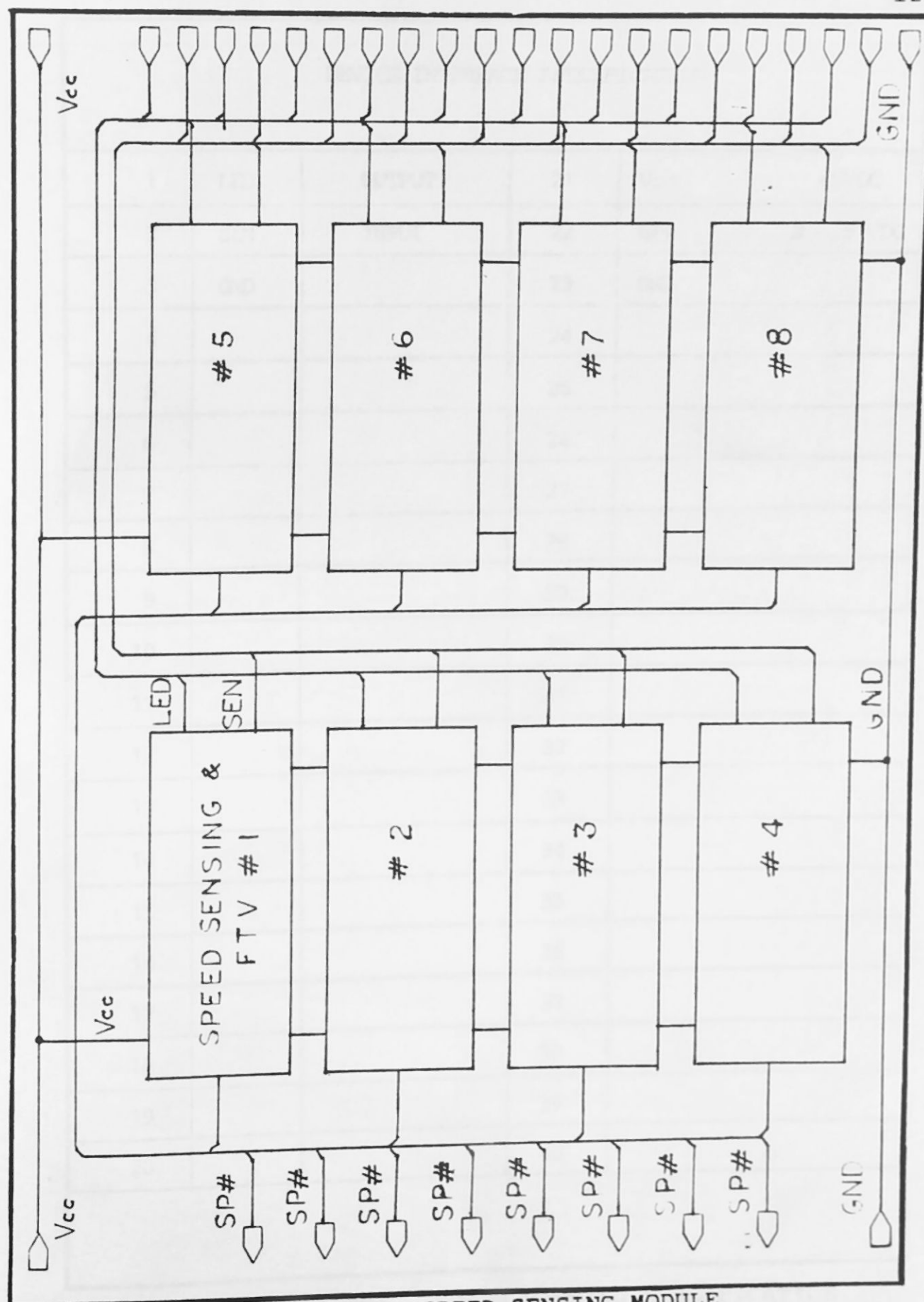


Figure A-19: SPEED SENSING MODULE

DEVICE INTERFACE SPECIFICATION				
1	LED	OUTPUT	21	Vcc +5VDC
2	SEN	INPUT	22	SP# $\emptyset - 5$ VDC
3	GND		23	GND
4			24	
5			25	
6			26	
7			27	
8			28	
9			29	
10			30	
11			31	
12			32	
13			33	
14			34	
15			35	
16			36	
17			37	
18			38	
19			39	
20			40	

Figure A-20: SPEED SENSING SPECIFICATION

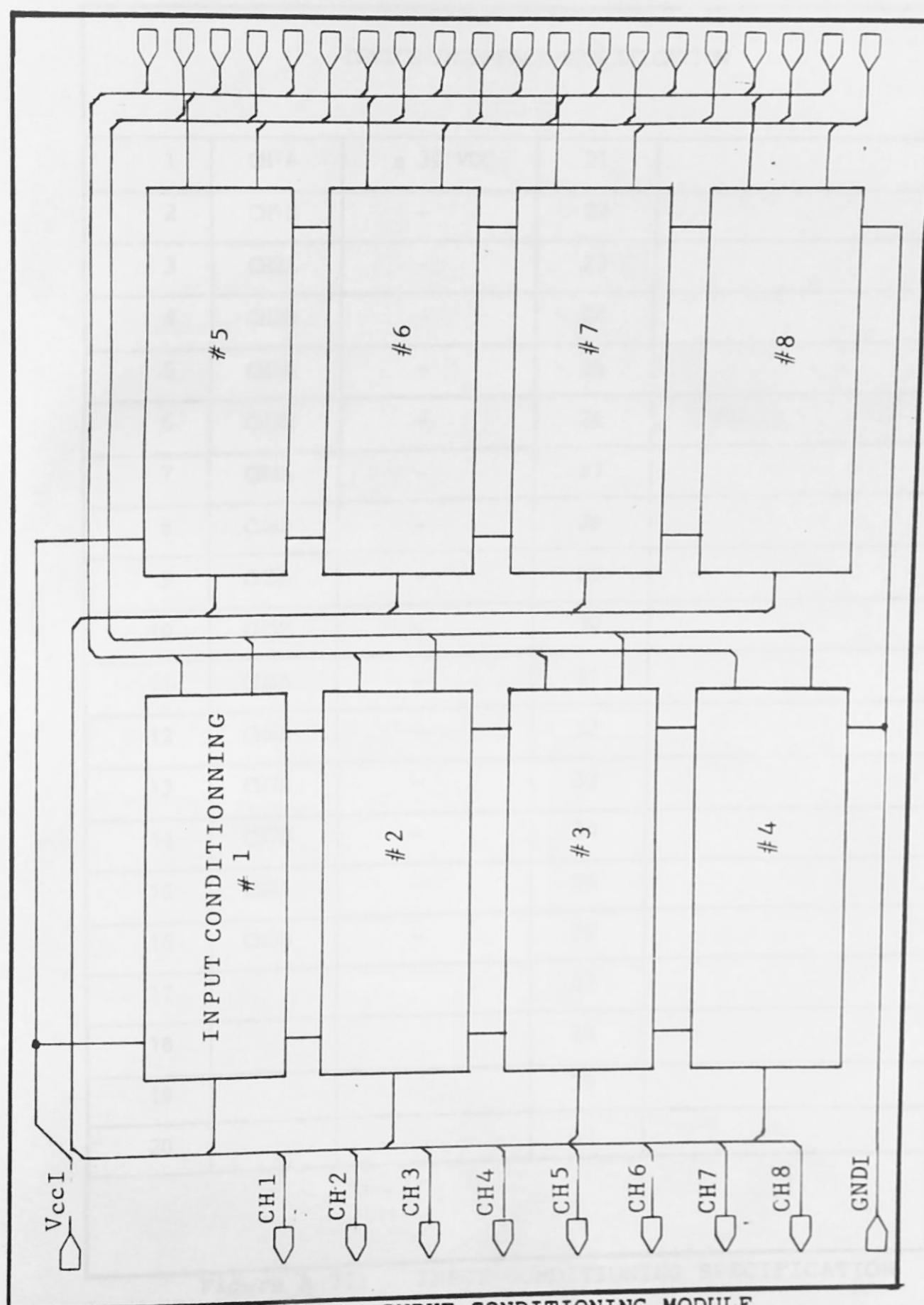


Figure A-21: INPUT CONDITIONING MODULE

DEVICE INTERFACE SPECIFICATION				
1	CH1A	± 30 VDC	21	
2	CH1B	-	22	
3	CH2A	-	23	
4	CH2B	-	24	
5	CH3A	-	25	
6	CH3B	-	26	
7	CH4A	-	27	
8	CH4B	-	28	
9	CH5A	-	29	
10	CH5B	-	30	
11	CH6A	-	31	
12	CH6B	-	32	
13	CH7A	-	33	
14	CH7B	-	34	
15	CH8A	-	35	
16	CH8B	-	36	
17			37	
18			38	
19			39	
20			40	

Figure A-22: INPUT CONDITIONING SPECIFICATION

REFERENCES NOT CITED

1. Sylvan, John, "Control software for factory automation," COMPUTER DESIGN, April 1983, pp. 119-126.
2. Zimmerman, Mark, "Floptran-IV : A Tiny Compiler," BYTE MAGAZINE, October 1980, pp. 196-228.
3. Grabol, Dan, "Real-Time Language For Industrial Microcomputer System," SYSTEM & SOFTWARE, April 1984, pp. 156-159.
4. P. M. Lewis II, D. J. Rosenkrantz, R. E. Sterns, *Compiler Design Theory*, Addison-Wesley, Reading, Massachusetts, The Systems Programming Series, 1978.
5. F.L. Bauer and J. Eickel, *Compiler Construction - An advanced Course*, Springer - Verlag, New York, Heidelberg, Berlin, 1976.
6. S. T. Allworth, *Introduction To Real Time Software Design*, Springer - Verlag NewYork Inc., 175 Fifth Avenue, NewYork, NY 10010, 1981.
7. Jeffrey R. Weber and Stephen Szcrecinski, *User's Handbook to the VIC-20 Computer*, Weber Systems, Inc., 8437 Mayfield Road, Cleverland, Ohio 44026, 1983.
8. Nick Hampshire, *Vic Revealed*, Hayden Book Co., Inc., Rochelle Park, New Jersey, 1982.